REPORT DOCUMENTATION PAGE

1

Form Approved OMB NO. 0704-0188

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
31-08-2014	MS Thesis	-

4. TITLE AND SUBTITLE
Mapping the Conjugate Gradient Algorithm onto High Performance Heterogeneous Computers

5a. CONTRACT NUMBER
W911NF-13-1-0133

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
206022

6. AUTHORS
Jamory Hawkins

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES
Jackson State University
1400 John R. Lynch Street

Jackson, MS 39217 -0129

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
62906-CS-REP.2

12. DISTRIBUTION AVAILIBILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT

ABSTRACT
Mapping scientific kernels onto high performance heterogeneous computers (HPHC) must comply with certain rules of thumb or heuristics. Previous research by Jackson State University's (JSU) HPHC research group has provided anecdotal evidence illustrating some of these rules/heuristics. The research highlighted by this thesis corroborates the credibility of these rules. In particular, four versions (two pairs) of a floating-point sparse

15. SUBJECT TERMS
High Performance Heterogeneous Computing

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Khalid Abed
UU	UU	UU	UU		19b. TELEPHONE NUMBER 601-979-3920

Report Title

Mapping the Conjugate Gradient Algorithm onto High Performance Heterogeneous Computers

ABSTRACT

ABSTRACT

Mapping scientific kernels onto high performance heterogeneous computers (HPHC) must comply with certain rules of thumb or heuristics. Previous research by Jackson State University's (JSU) HPHC research group has provided anecdotal evidence illustrating some of these rules/heuristics. The research highlighted by this thesis corroborates the credibility of these rules. In particular, four versions (two pairs) of a floating-point sparse matrix conjugate gradient (CG) iterative solver are presented. JSU's state-of-the-art HPHC utilizes general purpose processors (GPP) and heterogeneous computational hardware, in particular, a field programmable gate array (FPGA), to develop the CG kernels. The first version of the pair executes strictly on the GPP and the second uses both the GPP and FPGA to map the entire CG algorithm onto hardware. For the second pair, a refactored version of CG is used, which is statically analyzed to determine where the most computationally expensive operation occurs. This operation is the sparse matrix vector multiply (MVM) kernel. Based on this analysis, the software version of CG is refactored to call MVM as a subroutine. An FPGA version of the MVM algorithm is also developed and a static analysis of that algorithm suggests a speedup of the MVM kernel. All four version of CG are executed using a specially designed set of sparse matrices and the results demonstrate that adherence to the rules of thumb and heuristics when mapping scientific kernels onto a HPHC can lead to significant speedups.

MAPPING THE CONJUGATE GRADIENT ITERATIVE SOLVER ONTO HIGH PERFORMANCE HETEROGENEOUS COMPUTERS

by

Jamory D. Hawkins

A Thesis

Submitted to the Division of Graduate Studies
Jackson State University
In Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

May 2014

Major Subject: Computer Engineering

MAPPING THE CONJUGATE GRADIENT ITERATIVE SOLVER ONTO HIGH PERFORMANCE HETEROGENEOUS COMPUTERS

A Thesis

by

Jamory D. Hawkins

Approved:

Committee Chairperson Dr. Khalid Abed	Committee Member Dr. Gerald Morris
Committee Member Dr. Kamal Ali	Committee Member Dr. Gordon Skelton

Dean, Division of Graduate Studies
Dr. Dorris Robinson-Gardner

May 2014

To my family, friends, and those who believed in me, thank you for your love and support.

TABLE OF CONTENTS

CHAPTER Page

APPENDIX Page

LIST OF FIGURES

LIST OF SYMBOLS, ABBREVIATIONS, AND NOMENCLATURE

API Application Programming Interface

ASIC Application Specific Integrated Circuit

BITGEN Bit Generation Tool

CLB Configurable Logic Block

FPGA Field Programmable Gate Array

GPP General Purpose Processor

HDL Hardware Description Language

HLL High Level Language

HPC High-Performance Computing

HPHC High-Performance Heterogeneous Computer

I/O Input/Output

IP Intellectual Property

IPI Intellectual Property Interface

LUT Lookup Table

PAR Place and Route

PE Processing Element

PLD Programmable Logic Device

RC Reconfigurable Computer

SYNTH Syntheseis Tool

VHDL Very High Speed Integrated Circuit Hardware Description Language

ACKNOWLEDGMENTS

First and foremost I would like to thank Jesus Christ for leading me out of the darkness and blessing me with knowledge and wisdom. To my mother and grandparents, words cannot begin to comprehend how much your unwaivering support in my endeavors motivated me to grow stronger and go harder throughout the trials and tribulations that crossed my path. To my sisters, Shelnaka and Rashia, thank you for always believing and expecting the best of me. I would like to thank the faculty, staff, and administration at Jackson State University for creating a superior learning environment.

This work was supported in part by the U.S. Army Research Office under contract numbers W911NF-13-1-0133, Exploring High Performance Heterogeneous Computing via Hardware/Software Co-Design, and in part by the U.S. Army Engineer Research and Development Center. I would definitely like to thank my two advisors, Dr. Khalid Abed and Dr. Gerald Morris, for selecting me out of dozens of students to participate in this research program. I have learned from you tremendously both in the technical and professional aspect. I am proud to call you my mentors and without your support and guidance, I would not be the person I am today. To Mr. Anas Alfarra, Mr. Christopher Robinson, Mr. Michael Knight, and countless others that have been with me on this journey since day one, thank you for your companionship, more importantly, your friendship. Having someone to share your joy, pain, ups and downs, builds bonds and

lasting memories. Thank you for letting me lean on you when times got hard. I will forever be grateful for you.

ABSTRACT

Mapping scientific kernels onto high performance heterogeneous computers (HPHC) must comply with certain rules of thumb or heuristics. Previous research by Jackson State University's (JSU) HPHC research group has provided anecdotal evidence illustrating some of these rules/heuristics. The research highlighted by this thesis corroborates the credibility of these rules. In particular, four versions (two pairs) of a floating-point sparse matrix conjugate gradient (CG) iterative solver are presented. JSU's state-of-the-art HPHC utilizes general purpose processors (GPP) and heterogeneous computational hardware, in particular, a field programmable gate array (FPGA), to develop the CG kernels. The first version of the pair executes strictly on the GPP and the second uses both the GPP and FPGA to map the entire CG algorithm onto hardware. For the second pair, a refactored version of CG is used, which is statically analyzed to determine where the most computationally expensive operation occurs. This operation is the sparse matrix vector multiply (MVM) kernel. Based on this analysis, the software version of CG is refactored to call MVM as a subroutine. An FPGA version of the MVM algorithm is also developed and a static analysis of that algorithm suggests a speedup of the MVM kernel. All four version of CG are executed using a specially designed set of sparse matrices and the results demonstrate that adherence to the rules of thumb and heuristics when mapping scientific kernels onto a HPHC can lead to significant speedups.

CHAPTER 1

INTRODUCTION

1.1 History and Thesis Organization

High Performance Heterogeneous Computers (HPHC) have consistently delivered superior performance for their targeted application as opposed to their microprocessor based counterpart. Despite this significant speed-up, only a sparse number of applications were considerable to justify the design, development, and support costs of an HPHC (SRC Computers, LLC 2010). This is why mainstream supercomputing companies such as Cray and SGI have had HPHC offerings in the past but with fleeting commercial success, however, Seymour R. Cray's startup company, SRC Computers (SRC Computers, LLC 2010), has had some commercial success in the HPHC space (Jane's Information Group 2011). This success is partially attributable to their Carte compiler, which integrates their proprietary MAP processors and user-specific applications written in Fortran and/or C, into a single application executable code (SRC Computers Inc. 2014).

Previously, most complex systems in communications, military, and medical applications were simulated or modeled using floating-point data processing in C or MATLAB, which is a widely used program for solving algebraic and differential equations. However, the lack of support with field programmable gate array (FPGA) tool suites made floating-point arithmetic an unnappealing option for the FPGA designer,

thus, final implementations were nearly always performed using fixed-point or integer arithmetic (Parker 2009). With the recent advancements in FPGA technologies, companies such as Altera now offer single and double precision floating intellectual property (IP) cores fashioned specifically for high performance applications. Tools such as Altera's MegaWizard Plug-In Manager (Altera Corp. 2011) takes the hassle out of custom IP core manufacturing by automatically generating optimized floating-point IP cores within the FPGA itself. Such IPs have been successfully mapped onto HPHCs (Peay, Morris, and Abed 2011) and floating-point kernels targeting specific problem domains such as molecular dynamics have been demonstrated (Scrofano, Gokhale, Trouw, and Prasanna 2006; Herbordt, Khan, and Dean 2009), as have general-purpose kernels such as linear algebra routines (Zhuo and Prasanna 2005b; Zhuo and Prasanna 2005a; Wu, Dou, Lei, Zhou, Wang, and Jiang 2009). There has even been some success at mapping iterative solvers onto FPGAs (Roldao and Constatinides 2010; Morris and Prasanna 2006; Morris and Prasanna 2007).

For numerical and computational applications, these FPGA-based processors must satisfy a variety of heuristics and rules of thumb to achieve a speedup compared with their software counterparts. This thesis illuminates the challenging computational mapping process while simultaneously showing that such mappings can result in significant speedups. Its major focus is to demonstrate the veracity of the concept "the three P's" which expresses the crucial relationship among performance, pipelining, and parallelism, which is perhaps the most important mapping consideration (Morris and Abed 2013). In

addition, portions of this thesis is referenced from the paper which will be submitted to the International Conference on Engineering of Heterogeneous Systems and Algorithms (ERSA) (Hawkins, Alfarra, Morris, and Abed). This research paper is organized into five chapters. Chapter 1 begins with the introduction which identifies the problem and presents the significance of this research. Chapter 2 discusses previous work and literature that proves pivotal in the area of attaining a speedup on a HPHC. Chapter 3 discusses the tools, technology, and methodology used in conducting this research. Chapter 4 presents the results derived from this research and analyzes the platforms performance levels. Lastly, Chapter 5 concludes with future work related to achieving speedup with HPHCs.

1.2 Research Objective

The results and conclusions obtained from this research may be applied to various HPHC areas. By way of a sparse matrix conjugate gradient (CG) iterative solver, this paper illustrates some of the issues associated with mapping scientific kernels onto HPHCs. The CG method was chosen based on heuristics developed from earlier research(Rice, Abed, and Morris 2009; Morris and Abed 2013; Morris, McGruder, and Abed 2010). Furthermore, CG is a method that is frequently used in the real world and complex enough to deal with issues of mapping scientific kernels onto FPGAs.

In this research, four versions (two pairs) of CG were mapped on to a HPHC. The first version of the pair executes strictly on the general purpose processor (GPP) in JSUs state-of-the-art HPHC, and the second utilizes both the GPP and the heterogeneous

computational hardware, in particular, a field programmable gate array (FPGA). Not too surprisingly, the first hardware version does not perform very well because it does not consider the rules and heuristics that will be discussed in this thesis.

The second pair, which is a profiled version of CG, is statically analyzed to determine where the most computationally expensive operation occurs. This operation is the sparse matrix vector multiply (MVM) kernel. Based on this analysis, the software version of CG is refactored to call MVM as a subroutine. This software version is then profiled to determine the percentage of runtime spent in MVM. This factor is called the fraction that can be enhanced (f_e) in Amdahl's Law. A hardware (FPGA) version of the MVM algorithm is also developed and a static analysis of that algorithm suggests a speedup of the MVM kernel. This factor is called the speedup of the enhanced component (s_e) in Amdahl's Law.

The combination of these two factors (f_e and s_e) are used in Amdahl's Law to estimate the anticipated overall speed (s_o) up of the HPHC version of CG. The second HPHC version performs significantly better than the first because the rules and heuristics (to be described later) are followed, i.e., MVM was a great candidate for kernel mapping onto a HPHC to achieve an overall speedup of CG.

CHAPTER 2

LITERATURE REVIEW

This chapter discusses the necessary background information that serves as a platform for this research. This chapter is partitioned into 7 sections. The first gives rise to a particular type of HPHC as it examines the history and early mappings of FPGAs. Section two discusses heterogeneous computers (RC) and their physical architecture. Section three details the history and signficance of the conjugate gradient algorithm. The fourth section discusses matrix storage formats and their crucial importance in scientific research. Section five examines the types of matrices and their importance as well. Section six discusses in detail the HPHC target platform; the SRC-7. The seventh and final section of this chapter details Amhdal's Law which is an approach for calculating the overall speed up of a system when only part of the system has been enhanced.

2.1 FPGAs

2.1.1 History

Field programmable gate arrays are a programmable digital logic device (PLD). It was invented in 1984 by Xilinx co-founder Ross Freeman (Xilinx, Inc. 2006). At the time it was groundbreaking due to its ability to be dynamic in application. However, it operated at much slower speeds than Application-Specific Integrated Circuits (ASICs). It also used

more transistors than its fixed competitor. With the reduction in transistor cost, over time, FPGAs have become widespread in use.

2.1.2 Architecture

The physical architecture of an FPGA, as shown in Figure 2.1, consists of various components that are able to communicate via switches. These switches are directional and allow multi-point interconnection between components. Fixed logic blocks consist of, but are not limited to, the central processing unit (CPU), random access memory (RAM), and clock. Input/output (I/O) blocks allow for data to be made available to the FPGA as well as provide output from the application. Configurable logic blocks (CLBs) are structures that are used to emulate digital logic functions.

As shown in Figure 2.2, CLBs consist of multiple slices that are internally connected aside from the FPGA component routing mechanism. A deeper look into the slices reveals the components that allow for the FPGAs configurable design. Slices are comprised of multiple lookup tables (LUTs), flip-flops, multiplexers, and control. The CLB mimics a digital logic function set in place by the developer due largely in part to the lookup tables (LUTs) that lie within the slices. LUTs directly resemble digital logic using memory, as depicted in Figure 2.3.

At runtime, the FPGA is programmed with a configuration bitstream (the bitstream is viewed as data, but it is the driving force of the conversion of the FPGA). It defines the contents of LUTs, interconnections between components, and input or output

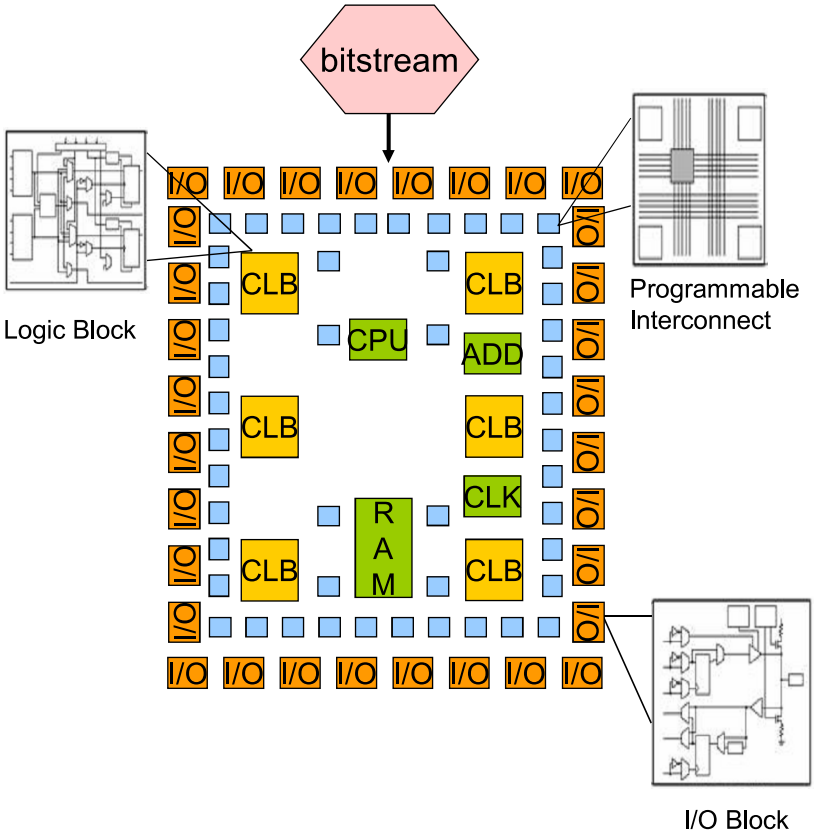

Figure 2.1 FPGA Architecture

(I/O). Hence, a different application implemented on an FPGA lies solely within the corresponding bitsream. The FPGA is thereby dynamic in functionality though it is limited in speed capability in comparison to its ASIC competitor. If a new application is to be developed, there is no need for “hard wiring” as in the historical process. Nonetheless, it is not to be assumed that generating the corresponding bitstream is a trivial task - for this now takes many steps and design considerations by the developer.

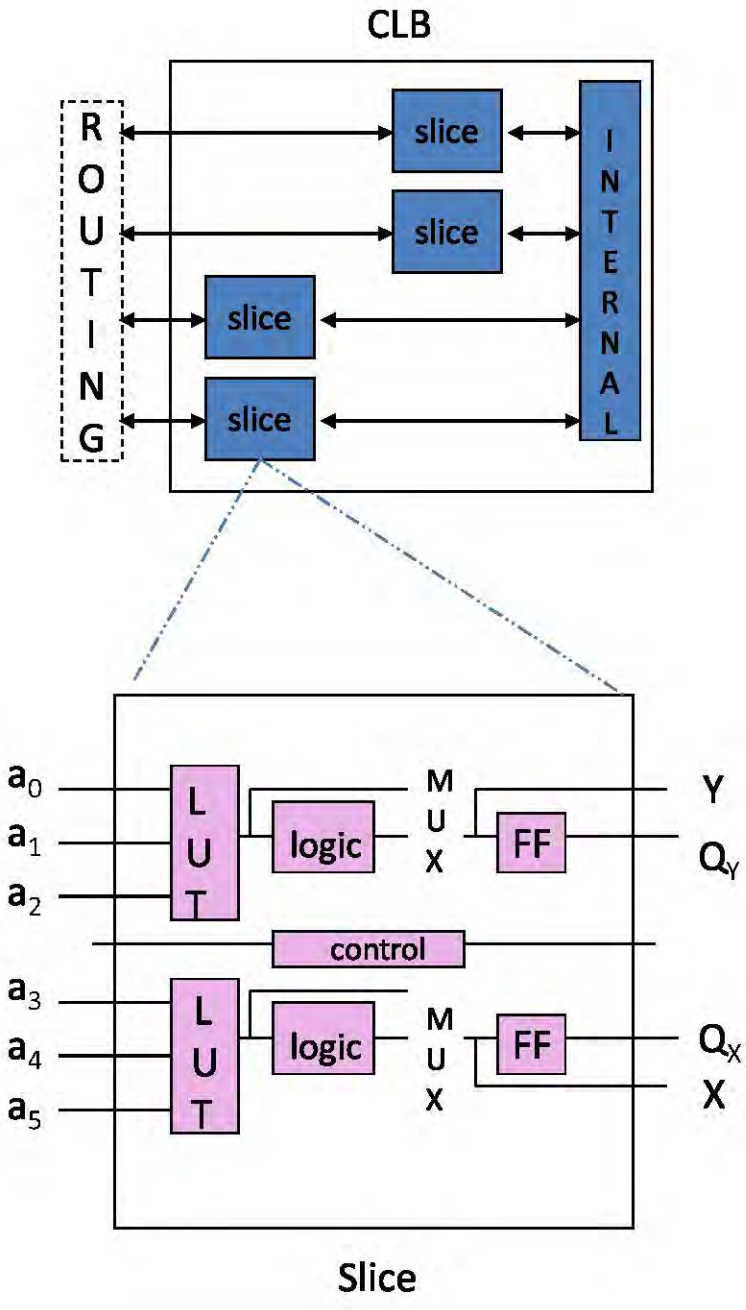

Figure 2.2 CLB architecture

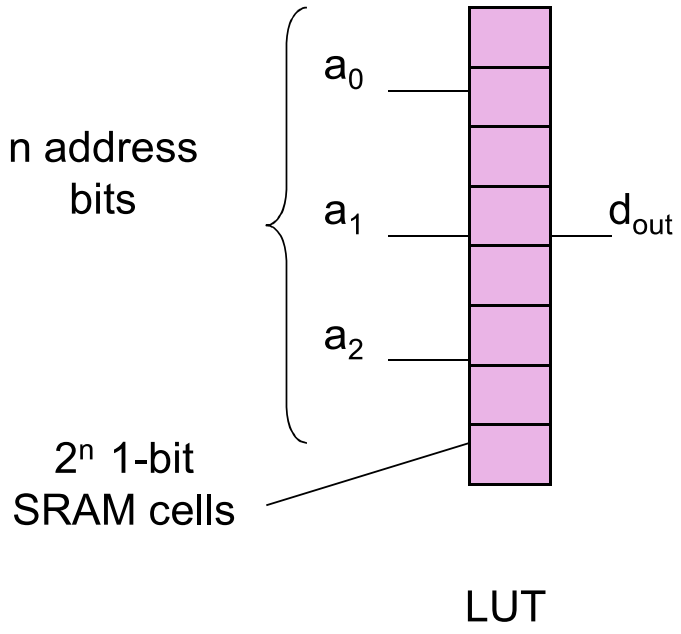

Figure 2.3 Lookup Table

2.1.3 FPGA Benefits and Applications

FPGAs have gained popularity as a platform for scientific computation applications. If inadequately designed, the platform may significantly decrease performance (Zhuo, Morris, and Prasanna 2007). Developers have also used the FPGAs to create significant increases over comparable platforms in application-specific integrated circuits (ASICs). For example, a 1800x increase in training speed has been obtained with Rankboost web search engines over its prior implementation using only a GPP (Xu, Cai, Gao, Zhang, and Hsu 2009). A 5.5 fold speedup in pair-wise computations was actualized by Nallamuthu et.al using the heterogeneous computing environment. FPGAs provide the potential for

high performance with less power consumption. Under the aforementioned development, a nine fold increase in the ratio of power to performance also gave an added benefit to using a heterogeneous environment involving an FPGA architecture (Nallamuthu, Smith, Hampton, Agarwal, and Alam 2010).

2.2 Heterogeneous Computers

Heterogeneous computing is a technology that encapsulates fixed and variable structures. It allows the developer to modify the architecture of a processing element. Unlike fixed hardware, the variable hardware is able to be adapted to a new platform in real time. The RC concept was invented in 1960 by Gerald Estrin. At the time, research and development efforts proved impractical. Developers would have to manually place modules and wire them according to the anticipated configuration. This lengthy approach was highly susceptible to human errors, and circuits would be limited in size. Presently, the manual process has been replaced with digital control switches that allow the variable structure to be configured through programming. Contemporary high performance heterogeneous computing (HPHC) clusters vary with its models and architectures. It may consist of multiple nodes of RCs to create parallelism in computation and task execution.

2.2.1 Application mapping

An application may be realized solely in software running on a GPP. This process usually involves code written in a high-level language that is compiled and linked with

system libraries to produce a binary executable. On a GPP with a single processor, each task is executed sequentially. Here, the developer's primary avenue of attaining a speedup is through optimizing the code. With an application placed on a heterogeneous computer, the fixed component generally calls for a GPP, and the variable component may be an FPGA. The application is developed and placed on an RC if deemed more profitable in terms of speedup. This concept is illustrated in Figure 2.4. It is important to analyze the potential of a computationally intensive kernel that is to be successfully placed on an FPGA. The developer strategically evaluates design considerations of the application. This is determined through various metrics that analyze its application size, algorithm, and memory-access patterns (Rice, Abed, and Morris 2009).

In order to profile the potential of an computational kernel that is to be placed onto the FPGA, the developer takes various design elements into consideration. In terms of runtime, the developer must analyze the kernel and determine its overall effect on the entire application. Its particular algorithm is evaluated as the developer must examine the kernel size in relation to space available on the FPGA. Concerning the entire RC layout, one must also consider memory access patterns. The bandwidth available for communication between the fixed and variable elements should be capable to transfer data without creating a significant increase in the overall runtime. If so, some of this time lost in transfer among elements may be alleviated through data reuse once it is placed onto the variable computing environment. This research considers a dot product for its computational kernel.

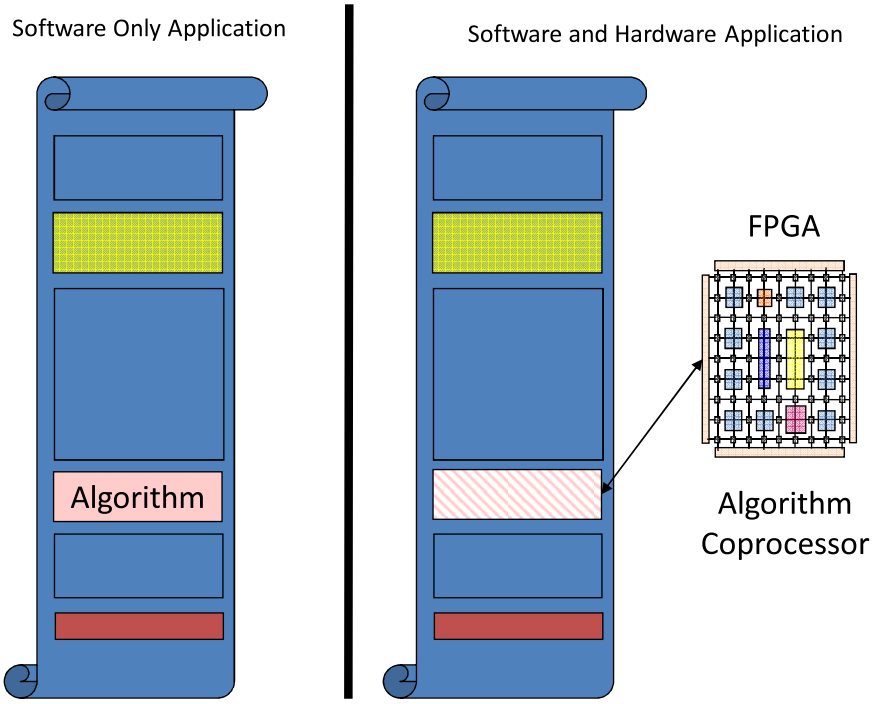

Figure 2.4 Application Mapping

2.3 Conjugate Gradient

The Conjugate Gradient (CG) method, which was discovered in 1952 by Hestens and Stiefel, is an algorithm typically used to solve large and sparse systems of linear equations, $A\mathbf{x} = \mathbf{b}$. The method was abandoned in the 1960s due to a lack of finite precision computational engines which caused the algorithm to diverge, thus, rendering it useless. In 1971, however, mathematician J.K. Reid indicated that CG, if observed

as an iterative method rather than an $\mathbf{n}$ step process, will ultimately converge with veritable accuracy (Paige and Saunders 1975). Today, the Conjugate Gradient method is acknowledged as one of the top 10 algorithms of the twentieth century (Van der Vorst 2000). It is the de facto standard algorithm for solving equations that involve symmetric positive definite (SPD) matrices (Morris 2013).

2.3.1 CG algorithm

This section discuss conjugate gradient as an iterative method to solve systems of linear equations

$$A\mathbf{x} = \mathbf{b} \quad \equiv \quad \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{2.1}$$

where $A \in \mathbb{R}^{n \times n}$ is a symmetric ($A = A^T$) positive definite ($\forall \mathbf{x} \neq 0, \mathbf{x}^T A\mathbf{x} > 0$) matrix, $\mathbf{x} \in \mathbb{R}^n$ is the unknown vector (our desired solution), and $\mathbf{b} \in \mathbb{R}^n$ is the constant vector.

It can also be shown that the solution $A\mathbf{x} = \mathbf{b}$ is the same as the $\mathbf{x}$ vector that minimizes the quadrtic function,

$$f(\mathbf{x}) = \frac{1}{2} \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

i.e., $\nabla f(\mathbf{x}) = 0 = A\mathbf{x} = \mathbf{b}$, which is also a linear combination of A-orthogonal(conjugate) vectors $\{\mathbf{p}_1, \mathbf{p}_2, \cdots, \mathbf{p}_n\}$ that forms a basis for $\mathbb{R}^n$.

$A\mathbf{x} = \mathbf{b}$ can also be expressed in Equation 2.2 as

$$\mathbf{x} = \sum_{i=1}^{n} \alpha_i \mathbf{p}_i. \tag{2.2}$$

Finding the A-orthogonal vectors is a non-trivial task; therefore, an iterative approach is used. In particular, it can be shown that each of these vectors correspond to a search direction, therefore, the iterative approach is in effect a gradient descent method that minimizes the quadratic function previously mentioned. After each step in the given direction there is a residual, $\mathbf{r}_k$, which in a sense indicates how far off the solution is (Morris 2013). It can be shown that these search directions are in a direction negative of the gradient at the starting point, hence, the term "*conjugate gradient*". Think of "zig-zagging" to the bottom of a bowl. Performing this type of gradient descent with infinite precision, the iteratively generated set of A-orthogonal vectors would correspond to a basis for $\mathbb{R}^n$ as depicted in Figure 2.5.

Via some algebraic manipulations this approach can be refined to a recurisve technique, however, one of the issues with this technique is that the residual contains components in the search direction that have already been taken. Remember, the next search direction must be A-orthogonal. It is necessary to remove these components from the residual in order to calculate a new search direction via the projection operater β which can be observed as $\beta_k = \dfrac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}$. Before discussing the final algorithm, one detail

15

1: **algorithm** PRIMITIVECG($A, \mathbf{x}, \mathbf{b}$)

2: Using current point, calculate residual, $\mathbf{r}_k \leftarrow \mathbf{b} - A\mathbf{x}_k$

3: Calculate next conjugate vector, $\mathbf{p}_{k+1} \leftarrow \mathbf{r}_k - \sum_{i \leq k} \beta_{ik} \mathbf{p}_i$

4: Descend to the next point $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_{k+1} \mathbf{p}_{k+1}$

5: If not the minimum point, goto 2

6: **end algorithm**

Figure 2.5 Primitive CG algorithm

that needs to be mentioned is the classical residual norm. This is a standard approach used in iterative methods such as CG, Jacobi, Newton-Raphson, etc.

After a further improvements, an optimized version of CG is realized in Figure 2.6. With this approach, an arbitrary starting point $\mathbf{x}_0$ was selected which then descended. The first A-orthogonal vector, $\mathbf{p}_1$, is a search direction opposite to the gradient as depicted in line 2, therefore, the initial search direction is also the same as the initial residual as illustrated on line 3. The next line is a criteria for forcing the algorithm to enter the loop. Line 5, as mentioned previously, is part of the equation for calculating the residual. Rather than continually computing this value, it is simply calculated one time. For line 6, the body of the loop requires the dot product of the current residual, $\mathbf{r}_k$, and the previous residual, $\mathbf{r}_{k-1}$. Instead of calculating two dot products within the loop, the dot product from the previous iteration was reused, i.e., $\mathbf{r}_0^T \mathbf{r}_0$ is a dot product. Line 7 creates the

16

1: **algorithm** OPTIMIZEDCG($A, \mathbf{x}, \mathbf{b}$)
2: $\mathbf{p}_1 \leftarrow \mathbf{b} - A\mathbf{x}_0$
3: $\mathbf{r}_0 \leftarrow \mathbf{p}_1$
4: $\Delta \leftarrow \epsilon + 1$
5: over$b_{\text{norm}} \leftarrow 1/\|\mathbf{b}\|$
6: $rTr_{\text{old}} \leftarrow \mathbf{r}_0^T \mathbf{r}_0$
7: $k \leftarrow 1$
8: **while** $(\Delta > \epsilon)$ .AND. $(k < k_{\text{max}})$ **do**
9: $\mathbf{v}_{ap} \leftarrow A\mathbf{p}_k$
10: $\alpha_k \leftarrow rTr_{\text{old}}/\mathbf{p}_k^T \mathbf{v}_{ap}$
11: $\mathbf{x}_k \leftarrow \mathbf{x}_{k-1} + \alpha_k \mathbf{p}_k$
12: $\mathbf{r}_k \leftarrow \mathbf{r}_{k-1} - \alpha_k \mathbf{v}_{ap}$
13: $rTr_{\text{new}} \leftarrow \mathbf{r}_k^T \mathbf{r}_k$
14: $\beta_k \leftarrow rTr_{\text{new}}/rTr_{\text{old}}$
15: $rTr_{\text{old}} \leftarrow rTr_{\text{new}}$
16: $\mathbf{p}_{k+1} \leftarrow \mathbf{r}_k + \beta_k \mathbf{p}_k$
17: $\Delta \leftarrow \|\mathbf{r}_k\| \cdot$ overb_{norm}
18: k_{++}
19: **end while**
20: **return** $(\mathbf{x}_{k-1})$
21: **end algorithm**

Figure 2.6 Optimized CG algorithm

iteration index, k, which is also part of the criteria for the while loop to stop infinite looping.

It can be shown that the matrix vector product is needed twice within CG, however, it is computed once on line 9 because it is an expensive $\mathbf{O}(n^2)$ algorithm. The next line computes the step size, α_k. along the direction of the CG. For line 11, the new approximation for $\mathbf{x}$ is calculated by descending in the conjugate search direction of the step size, i.e., $\mathbf{x}_{k-1} + \alpha_k \mathbf{p}_k$. Line 12, recursively computes the new residual. Then the dot product of the new residual, rTr_{new} is calculated, which will be used in subsequent computations. Line 14 computes the projection operator β which removes from the

residual all previous search directions. Since the dot product of the previous residual is no longer needed and to prevent calculating the dot product at the next iteration, line 15 retains the current residual for the next iteration. The new search direction can finally be calculated, by using the projection operator to remove the residual from all components along the previous conjugate search directions, i.e., $\mathbf{r}_k + \beta_k \mathbf{p}_k$. Line 17 computes the residual norm to check the algorithm for convergence. Line 18 simply increments the iteration index. If the algorithm has converged, the solution, $\mathbf{x}_{k-1}$, is returned.

This optimized version will become the blueprint for the four versions of CG as discussed in Chapter 3.

2.4 Sparse Matrix Storage Formats

According to J. Dongarra (Dongerra 2000), the efficiency of most iterative methods, such as CG, can be attributed to the performance of the matrix-vector multiply which correlates directly to the storage scheme used for the matrix. The reasoning behind these formats begins with the concept “sparse matrix”. A matrix is considered sparse if there are more zero than non-zero elements, in which case, it is more efficient to store only the non-zero elements because memory consumption can be reduced and performance can be increased by using a specialized sparse storage representation (B. Jacobs 2013). Sparse storage schemes allocate contiguous storage in memory for the nonzero elements, which in turn requires a scheme for knowing where the elements fit into the full matrix (Dongerra 2000), i.e., storing all non-zero elements of the matrix into a linear array and

providing auxiliary arrays to describe the locations of the non-zero elements in the original matrix (D. Dodson and Lewis 1991). Coordinate format (COO) and Compressed Sparse Row (CSR) are the primary sparse matrix representations used throughout this research and shall be discussed in depth.

2.4.1 Coordinate Format

Also known as the 'ijv' or 'triplet' format, COO form is the simplest method to construct a matrix. It allows for very fast conversion to and from CSR format for arithmetic and matrix vector operations (Scipy 2009). COO format uses the vectors **row**, **col**, and **val** to store the row indices, the column indices, and values of the matrix, respectfully. For example, a 5 x 5 matrix is depicted in Fig. 2.7.

$$\begin{matrix} 7 & 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 & 5 \\ 2 & 0 & 6 & 1 & 1 \\ 0 & 4 & 7 & 0 & 0 \\ 9 & 3 & 0 & 0 & 4 \end{matrix}$$

Figure 2.7 Simple 5 x 5 matrix

This simple matrix can be easily converted to COO format as shown in Fig. 2.8. Given an index, i, that walks each element of the three vectors $Row, Column, Value$, one can easily reconstruct the original matrix.

row	1	1	2	2	3	3	3	3	4	4	5	5	5
col	1	4	1	5	1	3	4	5	2	3	1	2	5
val	7	2	3	5	2	6	1	1	4	7	9	3	4

Figure 2.8 Example of COO format

2.4.2 Compressed Sparse Row

If a matrix has a small number of nonzero values, $n_z \ll n^2$, where n is the matrix order and n_z is the number of nonzeros, CSR format (Saad 2009) can reduce storage and computational requirements. A real matrix with $n = 1,024$ and $n_z = 4,096$, which requires 4MB in dense format, can be stored in CSR format in about 36KB. Furthermore, needless computations with zero values can be avoided. For a real matrix of order n with n_z nonzeros, CSR uses vectors $\mathbf{val}$, $\mathbf{col}$, and $\mathbf{ptr}$ to store only the nonzero values and to identify the associated row and column indices. Vector $\mathbf{val}$ is a real vector of length n_z containing the nonzero values obtained from a row-wise traversal of the matrix. The $\mathbf{col}$ integer vector is also of length n_z and contains the column index of each nonzero value, i.e., $(val_h = a_{ij}) \Rightarrow (col_h = j)$. The $\mathbf{ptr}$ integer vector is of length $n + 1$ and contains the index in $\mathbf{val}$ where each matrix row starts. For example, the first nonzero element of matrix row m is found at index ptr_m of $\mathbf{val}$. By convention, $ptr_{n+1} \equiv n_z + 1$. Notice that $(a_{ij}) \Rightarrow (ptr_i \leq j < ptr_{i+1})$ for all i. An example, depicting the CSR representation of an order $n = 4$ sparse matrix, is shown in Fig. 2.9. Consider $ptr_3 = 5$; this indicates that row

$$\begin{bmatrix} 8 & 0 & 2 & 3 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 6 & 1 \\ 0 & 4 & 0 & 7 \end{bmatrix} \quad \begin{matrix} n{=}4 \\ \Longleftrightarrow \\ n_z{=}8 \end{matrix}$$

	1	2	3	4	5	6	7	8
val	8	2	3	5	6	1	4	7
col	1	3	4	2	3	4	2	4
ptr	1	4	5	7	9			

Figure 2.9 Example of CSR format

3 of the matrix begins at index 5 of **val** and **col**. Notice that $val_5 = 6$ and that $col_5 = 3$. Thus, $a_{3,3} = 6$ as in the dense representation of the matrix.

2.4.3 Aligned CSR Format

For the CG processor described in this paper, exactly $k = 8$ values are read from **val** and **col** during each clock cycle. This is accomplished by striping these vectors across multiple memory banks. To avoid large multiplexers and the associated degradation in performance, the matrix rows are simply padded with zeros (as needed) to ensure they have an exact multiple of k values. The **ptr** vector is modified to express indices in terms of k-sized groups of data, and the term kn_z (analogous to n_z) is the number of k-groups. This entire process, which is carried out by the software module when it marshals the data for the FPGA-based processor, is known as "k-alignment." This k-alignment process produces scalar kn_z and vectors **kval**, **kcol**, and **kptr**. As before, $kptr_{n+1} \equiv kn_z + 1$.

Perhaps the idea is better understood by way of an example. If one assumes that $k = 2$, then the sparse matrix represented in Fig. 2.9 could be represented in k-aligned CSR format as shown in Fig. 2.10. Each bracketed k-group of data represents the contents at a

given index across a striped data set. Consider $kptr_4 = 5$; this indicates that matrix row 4 begins at index 5 of striped memory banks **kval** and **kcol**. Notice $kval_5 = [4, 7]$ (2 banks,

	1	2	3	4	5
kval	$[8, 2]$	$[3, 0]$	$[5, 0]$	$[6, 1]$	$[4, 7]$
kcol	$[1, 3]$	$[4, 0]$	$[2, 0]$	$[3, 4]$	$[2, 4]$
kptr	1	3	4	5	6

Figure 2.10 Example of aligned CSR format ($k = 2, kn_z = 5$)

2 values) and that $kcol_5 = [2, 4]$ (2 banks, 2 indices). Thus, as in the dense matrix of Fig. 2.9, $a_{4,2} = 4$ and $a_{4,4} = 7$.

2.5 Test Matrices

During this research it became increasingly apparent that the complexity of the matrix itself played a significant role when analyzing the number of iterations and convergence rate. Conditioning sparse matrices into a form more suitable for numerical solution, or preconditioning (Axelsson 1996), resulted in rapidly converging solutions using the software version of CG, thereby rendering the speedup which had hoped to be obtained via the fully pipelined and parallelized hardware version counterproductive. Though preconditioning is the key to a successful iterative solver such as conjugate gradient (Song), the following section will discuss the ill-conditioned(non-preconditioned) matrices

employed that greatly challenge CG in addition to obtaining a significant speedup via a HPHC.

2.5.1 UFL Sparse Matrix Collection

The University of Florida Sparse Matrix Collection, managed by Dr. Tim Davis and Dr. Yifan Hu, is a large and actively growing set of sparse matrices widely used by the numerical linear algebra community for the development and performance evaluation of sparse matrix algorithms such as conjugate gradient. The collection covers a vast spectrum of domains ranging from mathematics and physics, to civil engineering and computer science. With over 2547 matrices, UFL's Sparse Matrix Collection boasts its largest matrix as having a dimension of 118 million with almost 2 billion nonzero entries (Davis and Hu 2011). UFL provides a Java program(UFgui) for browsing and downloading matrices in Matrix Market format and the complexity of the matrix itself can be manipulated by the user. Included with this software is an option for rendering images that correlate to the matrix generated by the user.

2.6 SRC-7

2.6.1 Target HPHC

The Jackson State University research group's HPHC, SRC-7, has two 3.0GHz Intel Xeon processors with a 16K L1 cache, 2MB L2 cache, and 6GB RAM. The MAP Series H processor contains two Altera EP2S180F1508C3 FPGAs running at 150MHz. It has

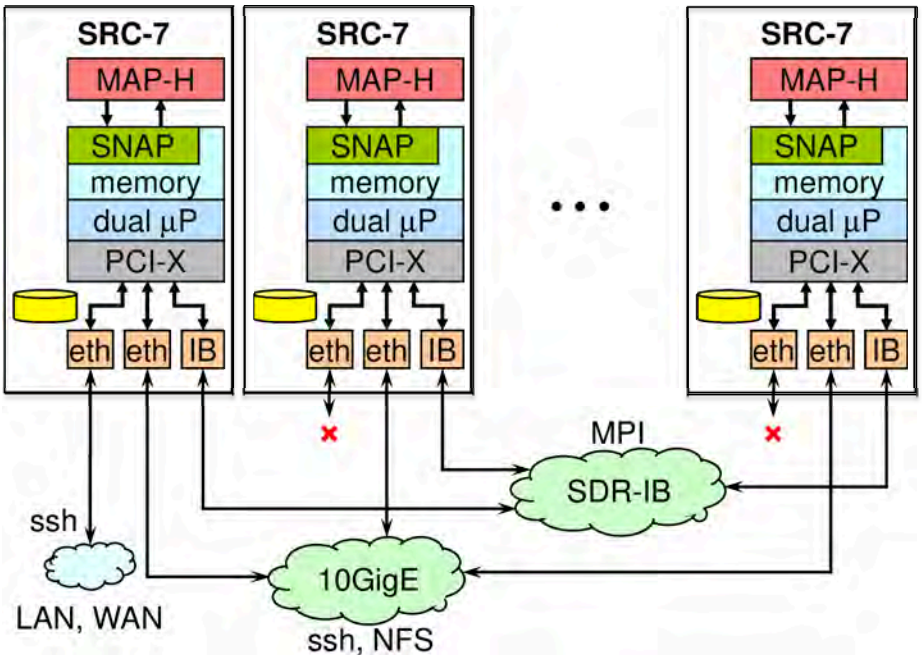

Figure 2.11 SRC-7 HPHC architecture

two 1GB banks of global common memory (GCM), which are used to marshal large data sets from the Xeon processor to the MAP processor and to return large data sets from the MAP to the Xeon. The Xeons access GCM via API calls, and the FPGAs access the GCM via DMA. GCM is the primary mechanism by which the GPP and FPGA interchange large data sets, i.e., GCM is used for bulk data transfers (as opposed to individual array elements). GCM can be accessed by the GPP via API calls and by the FPGA via direct memory access (DMA).

There are 16 banks of on on-board memory (OBM) associated with the FPGAs. Each bank contains over 500K 64-bit words and can be separately addressed. This allows the

MAP processor to simultaneously access up to 16 words per clock cycle. OBM is the primary mechanism by which large data sets can be striped for parallel consumption by the FPGA, i.e., up to 16 values can be read from OBM each FPGA clock cycle. DMA can be used to move data between the OBM and Xeon memory, but the use of GCM is more efficient. The MAP processor is connected to the Xeon GPP motherboard through SRC's proprietary SNAP D interface, which is mapped into the Xeon memory subsystem.

The MAP processor has a streaming DMA capability and an inter/intra-FPGA streaming capability that allows computation to overlap communication and facilitates parallelism within the MAP processor. Multiple configurable single-port BRAMs are located within the FPGA fabric. The BRAMs are typically used to hold arrays that are accessed serially. A single value can be read from each BRAM array during a given clock cycle.

2.6.2 HLL development flow

Vendors such as Mentor Graphics and Altera have created compilers, DK Design Suite (Mentor Graphics 2010) and Carte C (SRC Computers Inc. 2014), respectively, that takes the hassle out employing HDL-based hardware design. These HLL-to-HDL compilers allow scientists and engineers to program HPHCs using HLLs like C or Fortran, which in the past, was limited to only using complex HDLs. HLL-to-HDL compilers support enhanced language features such as pipelined loops, parallel code blocks, synchronization

primitives, and intellectual property interface (IPI) calls to access vendor or user IP cores. Figure 2.12 depicts an HLL development flow for HPHC-based applications.

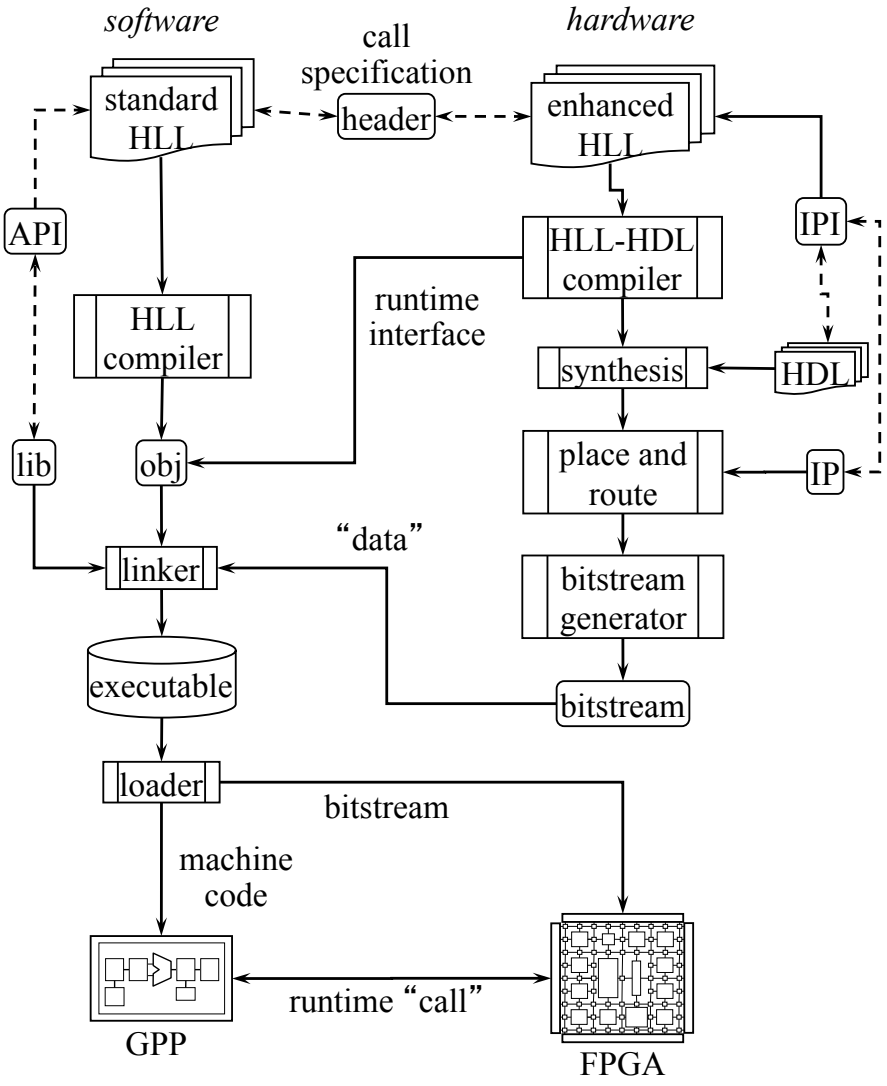

Figure 2.12 HLL development flow

The application is partitioned into software modules, which are targeted for execution on the GPP(s), and hardware modules, which are targeted for execution on the FPGA(s). Developers can also integrate custom HDL-based IP cores into their design. During development, the HLL software module is compiled with a standard HLL compiler to produce object files. The linker uses the object files (including the hardware module runtime interface code emitted by the HLL-to-HDL compiler), library files, and the configuration bitstream (treated as data by the linker) to produce a binary executable. The enhanced HLL hardware module is ingested by the HLL-to-HDL compiler, which emits HDL and the runtime interface code previously mentioned. The HDL is used by the synthesis tool to produce a netlist. The netlist, which topologically details the low-level device instances and connections needed to implement the logic circuit, is fed into the place and route tool. As suggested by the name, the place and route tool determines an optimal way to implement an instance of the netlist subject to a set of constraints, e.g., area, timing.

The output of place and route is fed to the bitstream generation tool to produce an FPGA configuration bitstream. The bitstream, when loaded onto the FPGA, configures the programmable logic, interconnection network, and I/O of the FPGA such that it implements the logic design representing the computational kernel. From the viewpoint of the software module, as suggested by the header file block at the top of figure 2.12, the hardware module “looks like” a parameterized subroutine call. From the viewpoint of the hardware module HLL code, IP cores also look like subroutine calls via the IPI. At

execution time, the configuration bitstream is extracted from the binary executable and loaded onto the FPGA. The G PP then executes the machine code instructions and “calls” the FPGA-based kernel as needed.

2.7 Amdahl’s Law

Gene Amdahl, a Norwegian American computer architect who has worked extensively on mainframe computing at IBM and later his own companies, is best known for formulating Amdahl’s law, which states a fundamental limitation of parallel computing (IEEE 2014). Amdahl’s law or Amdahl’s argument, is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors, however, Amdahl’s law also encompasses the domain of RC systems as well (Amdahl 1967).

Amdahl’s law is a model for the relationship between the expected speedup of parallelized implementations of an algorithm relative to the serial algorithm, under the assumption that the problem size remains the same when parallelized. Equation 2.3 gives the formula for Amdahl’s law.

$$s_o = \frac{1}{1 - f_e + f_e/s_e}, \tag{2.3}$$

where s_o is the overall speedup, f_e is the fraction of the system to be enhanced, and s_e is the speedup of the portion to be enhanced. For example, if 75 percent, i.e., the fraction to be enhanced, of an algorithm can be parallelized, and 25 percent is sequential, according

to Amdahl's law, the overall speedup which can be achieved on three processors, i.e., the portion to be enhanced, is 1/((1-.75) + (.75/3)) = 2. This means that the overall speedup, s_o, the program can gain is twice as fast on three processors than on one. Amdahl's law is used within this research to statically analyze the overall speedup of the monolithic and tuned versions of CG. The details are discussed in Chapter 3.

CHAPTER 3

METHODOLOGY

3.1 Overview

High performance heterogeneous computers that employ field programmable gate arrays (FPGAs) as computational elements are known as high performance heterogeneous computers (HPHCs). For floating-point applications, these FPGA-based processors must satisfy a variety of heuristics and rules of thumb to achieve a speedup compared with their software counterparts. By way of a sparse matrix conjugate gradient iterative solver, the following sections illustrates the methods associated with mapping floating-point kernels onto HPHCs.

It is worthwhile looking at some of the rules of thumb and heuristics used to determine if CG is a good mapping candidate. These have been developed over the past several years based on Jackson State University's HPHC-oriented research and are described in some detail in (Rice, Pace, Gates, Morris, and Abed 2008; Justin L. Rice and Morris 2009). Considerations include: (a) the three p's, (b) expected resource utilization, (c) control/memory vs. compute intensive, (d) monolithicity, (e) available bandwidth, (f) opportunities for data reuse, (g) algorithm design stability, (h) algorithm efficiency, and (i) memory access patterns. Section 3.2 gives more details about these design considerations relative to CG.

There are currently two pairs (monolithic and "tuned") of the floating-point sparse matrix CG iterative solver in reference with this research. The two monolothic versions blindly mapped the *entire* CG algorithm. The first version executes strictly on software, i.e., the GPP. The second utilizes both the GPP and the SRC-7 HPHC. Section 3.4 details the monolothic versions while also giving an overview of the pitfalls associated with CG when the rules and heurisitcs of mapping floating-point kernels onto HPHCs are not considered. The "tuned" pair are refactored versions of CG. CG was statically analyzed to determine where the most computational expensive operations occur, which is MVM. Through the use of a system call, CG was redeveloped to initiate MVM as a seperate subroutine. Section 3.5 discusses both the software and hardware versions of the profiled CG algorithm, a detailed discussion of MVM, and the analytics used for the theoretical overall speedup via Amdahl's Law.

3.2 The JSU-team rules and heuristics

3.2.1 The three p's

The three p's, which expresses the crucial relationship among performance, pipelining, and parallelism, is perhaps the most important mapping consideration (Morris and Abed 2013). The multiplicative effect of both pipelining and parallelizing is suggested by Eq. 3.1:

$$performance \propto pipelined \times parallelized \tag{3.1}$$

Given the order-of-magnitude clock speed advantage of GPPs over FPGAs, a candidate FPGA module must be able to be both pipelined and parallelized if a performance improvement is to be realized.

3.2.2 Expected resource utilization

Another important consideration is the expected FPGA resource utilization of the candidate module. Since floating-point IP cores can be quite large, the developer needs to determine if the candidate will even fit on the FPGA. The developer also needs to consider the needed local memory capacity, number of simultaneous memory accesses, anticipated clock rate in the light of complex routing, etc. In the case of the CG solver, the limiting factor was the amount of BRAM in the FPGA fabric, with the number of OBM banks running a close second. Despite these limitation an 8-wide parallel CG data path capable of handling sparse matrices up to order, $n = 8$K, with up to approximately 4M nonzero entries was constructed.

3.2.3 Control/memory vs. compute intensive

It is also imperative to consider whether an algorithm is control/memory intensive or compute intensive. The control aspect is similar to the branching problem in a GPP, and the memory aspect is similar to a GPP where accessing memory data takes a considerable amount of time compared with arithmetic operations. Harkins, et al., (Harkins, El-Ghazawi, El-Araby, and Huang 2005) illustrate the importance of this concept when they show that sorting algorithms do not perform very well on an HPHC.

The CG processor employs design features that minimize the number of control clauses, e.g., the use of k-aligned CSR format. Furthermore, unlike a GPP memory hierarchy, the HPHC memory organization does not penalize irregular memory accesses (Abed and Morris 2009). Therefore, on an FPGA, CG appears to be primarily compute intensive.

3.2.4 Monolithicity

If the candidate FPGA module contains procedure calls (is not monolithic), then these calls have to be inlined or the module cannot be considered as a viable candidate (hardware cannot "call" hardware). Obviously, the ability to inline is impacted by the available FPGA resources. In the case of the CG processor, the Carte compiler provided IPI "calls" for some of the lower level "routines" such as square root. Therefore, the first CG kernel pair was effectively monolithic and suitable for mapping onto an FPGA. In addition, a second refactored pair of CG kernels had a software subroutine that called MVM which is also monolithic.

3.2.5 Available bandwidth

The GPP to FPGA bandwidth also deserves attention. Obviously, the FPGA memory access and processing time should be less than the GPP memory access and processing time. According to Herbordt, et al., (Herbordt, Court, Gu, Sukhwani, Conti, Model, and DiSabello 2007), when they discuss latency hiding, a design should try to overlap computation with communication. In the CG processor a reduction in the bandwidth

requirement was acheived by marshaling the data into GCM banks and then using DMA to load the OBM and BRAM memory used during the FPGA-based computation.

3.2.6 Opportunities for data reuse

Algorithms that have a significant potential for data reuse may be suitable FPGA module candidates. In the case of the CG iterative solver, the matrix A and vector $\mathbf{b}$ are reused during every iteration. This had the effect of amortizing the transfer costs across all iterations.

3.2.7 Algorithm design stability

Since mapping an algorithm to an FPGA is not the easiest of tasks, it is imperative to make sure that the algorithm is as stable as possible. If the algorithm is altered while in the midst of a hardware implementation process, one could easily discover that the new algorithm no longer fits onto the FPGA, or that it can no longer deliver on the promised speedup. The CG iterative solver has been around for over six decades; obviously the authors did not anticipate any algorithm design modifications.

3.2.8 Algorithm efficiency

Another application design consideration is to make sure an efficient algorithm is employed. For example, Cramer's rule, which has exponential complexity, $O((n+1)!)$, might run faster if implemented on an FPGA. However, Gaussian elimination, with complexity, $O(n^3)$, is a much more efficient algorithm. In this case, one would use a

software solution rather than map the inefficient algorithm onto an FPGA. In the case of the CG algorithm, Section 2.3 identifies CG as being one of the most efficient algorithms for solving large sparse systems of linear equations.

3.2.9 Memory access patterns

Another design consideration is the memory access pattern. Unlike a GPP memory hierarchy, the HPHC memory organization does not penalize irregular memory accesses (Abed and Morris 2009). Since the CG processor uses a sparse matrix, it clearly demonstrates an irregular memory access pattern and is a good mapping candidate.

3.3 High-level design

3.3.1 CG

The high-level design for the CG solver is shown in Fig. 3.1. It consists of four major components: a main routine and matrix support libraries; several strictly diagonally dominant sparse matrices, $A_1 \ldots A_m$; the software or hardware (FPGA-based) CG solver; and the output result and statistics files, $\mathbf{x}_1 \ldots \mathbf{x}_m$ and $\mathbf{\Theta}_1 \ldots \mathbf{\Theta}_m$. The $\mathbf{b}_i$ vectors are shown as inputs, but for the experiments in this research they are generated from a known $\mathbf{x}$ vector at runtime. The main routine is a driver program, which essentially measures how long it takes for CG to solve each set of equations. The coordinate-format matrices are read in using the Matrix Market I/O library (NIST 2004) and converted to CSR format using Saad's SPARSKIT library (Saad 2009). The software CG kernel implementation

35

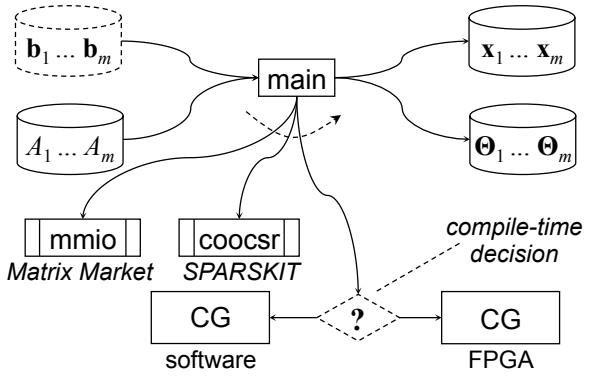

Figure 3.1 High-level CG design

is based on the algorithm shown in Figure 2.6, and the FPGA-based CG kernel will be described in Section 3.4.

A compile-time decision selects either the software or FPGA-based version of CG. At runtime, main reads in each coordinate-format matrix, converts it to CSR format, and uses a known $\mathbf{x}$ vector to generate $\mathbf{b}$. It then invokes the selected CG kernel sending matrix, A ($\mathbf{val}, \mathbf{col},$ and $\mathbf{ptr}$), starting point $\mathbf{x^{(0)}}$, and constant vector $\mathbf{b}$. After convergence, CG stores the result and returns. The main routine writes the solution to the results file; it also writes the input matrix name, number of iterations, and wall clock execution time to the statistics file and then terminates.

3.3.2 MVM

To recap, there were four CG algorithms that were mapped onto the HPHC. The first pair was a monolithic version which comprised of a software and hardware version. The second pair was ”tuned” and also had a software and hardware version, however, this

kernel involved a subroutine call that implemented MVM; software or hardware. In other words, the high-level design for the ”tuned” CG solver is very similar to Figure 3.1, however, in this kernel CG invokes MVM to either compute the software or hardware version as illustrated in Figure 3.2.

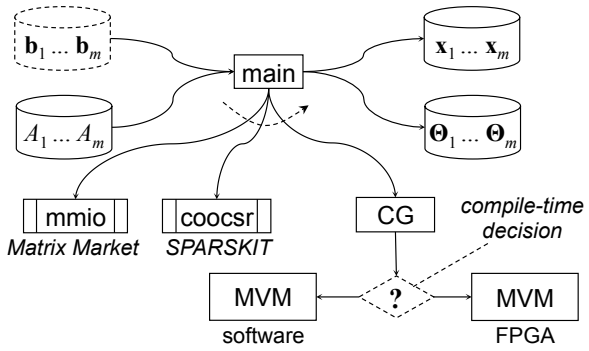

Figure 3.2 High-level MVM design

3.4 Monolithic version

3.4.1 Software

As stated in Section 1, our research began by blindly mapping the entire CG algorithm onto hardware without any regard to the heuristics that previous research has shown to highly impact performance. The monolithic CG algorithm that was executed in software, i.e., the GPP, is essentially the algorithm depicted in Figure 2.6.

3.4.2 Hardware

The monolithic hardware version of CG is shown in Figure 3.3.

1: **algorithm** CGHW($\mathbf{kval}, \mathbf{kcol}, \mathbf{kptr}, \mathbf{b}, \mathbf{b2}, \mathbf{n}, \mathbf{knz}, \mathbf{x}$)
2: **parBegin** // only two GCM banks so
3: BUF_DMA$_{\text{GCM}_1:\text{OBM}}$ ($\mathbf{kval}$, stripe-8)
4: BUF_DMA$_{\text{GCM}_2:\text{OBM}}$ ($\mathbf{kcol}$, stripe-8)
5: **parEnd**
6: **parBegin** // parallel DMA limited to 2
7: STREAM_DMA$_{\text{GCM}_1:\text{BRAM}}$ ($\mathbf{kptr}$)
8: STREAM_DMA$_{\text{GCM}_2:\text{BRAM}}$ ($\mathbf{b}$)
9: **parEnd**
10: **for** i **in** $[1, n]$ **do** // set up for repeat loop
11: $\mathbf{x}_i \leftarrow 0$ // x = 0
12: $\mathbf{p}_i \leftarrow \mathbf{r}_i \leftarrow \mathbf{b}_i$ // therefore p = b
13: $\mathbf{MAC}(\mathbf{r}_i, \mathbf{r}_i, \mathbf{rTrold})$ // calculate the dot product
14: **end for**
15: $\mathbf{d} \leftarrow 0$ // loop index
16: **repeat**
17: $\mathbf{p1} \leftarrow \ldots \leftarrow \mathbf{p11} \leftarrow \mathbf{p}$ // 11 copies of p vector
18: **parBegin** // compute v = Ap
$\mathbf{p}_1$: // feed values stream
19: **for** i **in** $[1, kn_z]$ **do**
20: $\mathbf{a} \leftarrow (a1 \cdots a8)$ stripe-8 from $kval_i$
21: $\mathbf{j} \leftarrow (j1 \cdots j8)$ stripe-8 from $kcol_i$
22: $\mathbf{p} \leftarrow (p1_{j1} \cdots p8_{j8})$
23: $V_{\text{FIFO}} \leftarrow$ dot8Tree $(\mathbf{a}, \mathbf{p})$
24: **end for**
$\mathbf{p}_2$: // feed counts stream
25: **for** i **in** $[1, n]$ **do**
26: $C_{\text{FIFO}} \leftarrow kptr_{i+1} - kptr_i$
27: **end for**
$\mathbf{p}_3$: // streaming accumulator
28: $S_{\text{FIFO}} \leftarrow \sum_{\text{STREAM}}(V_{\text{FIFO}}, C_{\text{FIFO}})$
$\mathbf{p}_4$: // the dotN's
29: **for** i **in** $[1, n]$ **do**
30: $v_i \leftarrow S_{\text{FIFO}}$
31: $\mathbf{MAC}(\mathbf{v}_i, \mathbf{p9}_i, \mathbf{pTv})$
32: **end for**
33: **parEnd**
34: $\alpha \leftarrow \mathbf{rTrold}/\mathbf{pTv}$ // step size
35: **for** i **in** $[1, n]$ **do**
36: $\mathbf{x}_i \leftarrow \mathbf{x}_i + \alpha\mathbf{p10}_i$ // next point
37: $\mathbf{r}_i \leftarrow \mathbf{r}_i - \alpha\mathbf{v}_i$ // residual
38: $\mathbf{MAC}(\mathbf{r}_i, \mathbf{r}_i, \mathbf{rTrnew})$ // new dot product
39: **end for**
40: $\beta \leftarrow \mathbf{rTrnew}/\mathbf{rTrold}$ // projection operator
41: $\mathbf{rTrold} \leftarrow \mathbf{rTrnew}$ // for next iteration
42: **for** i **in** $[1, n]$ **do**
43: $\mathbf{p}_i \leftarrow \mathbf{r}_i + \beta\mathbf{p11}_i$ // new search direction
44: **end for**
45: $\mathbf{r2b2} \leftarrow \sqrt{(rTrold)}\mathbf{b2}$ // residual norm
46: $\mathbf{d} \leftarrow \mathbf{d} + \mathbf{1}$ // next iteration
47: **until** $(r2b2 \leq \epsilon)$.OR. $(d > d_{\max})$
48: STREAM_DMA$_{\text{BRAM:GCM}_2}$ ($\mathbf{x}$) // all done, stream x back to GCM
49: **end algorithm**

Figure 3.3 monolithic hardware version of CG

39

The loops on line 35 and 42 are serialized loops, therefore, they cannot be done in parallel with the dot product, i.e., line 37. Consequently, these serial loops will incur costly computational penalties that significantly decreases the speedup. A block diagram is also presented here to further illustrate the algorithm.

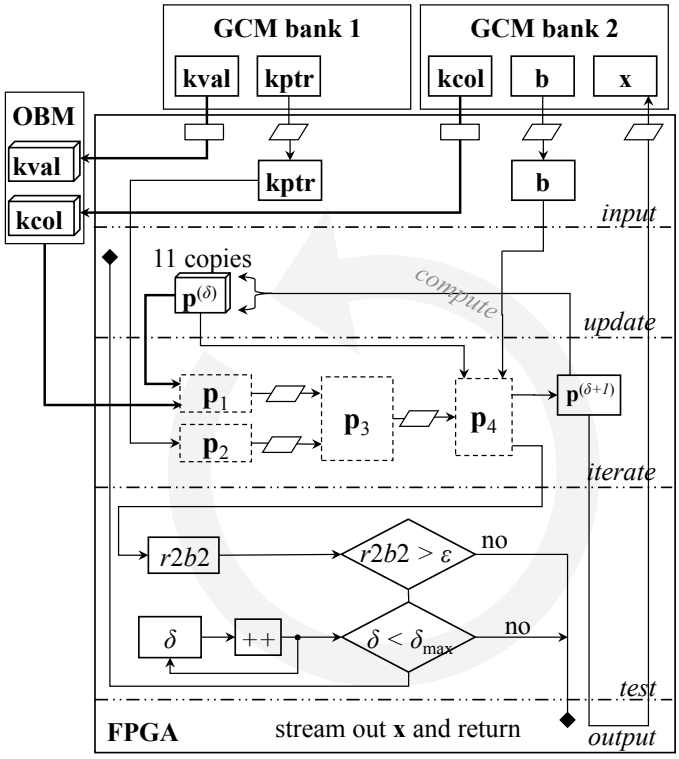

Figure 3.4 CG hardware process

3.5 "Tuned" version

3.5.1 Software

This algorithm is essentially the same as the discussion held in the previous section, however, this software features a subroutine call to MVM as shown in Figure 3.5. A static

1: **algorithm** CGSW($A, \mathbf{x}, \mathbf{b}$)
2: $\mathbf{p}_1 \leftarrow \mathbf{b} - A\mathbf{x}_0$
3: $\mathbf{r}_0 \leftarrow \mathbf{p}_1$
4: $\Delta \leftarrow \epsilon + 1$
5: ${}_{\text{over}}b_{\text{norm}} \leftarrow 1/\|\mathbf{b}\|$
6: $rTr_{\text{old}} \leftarrow \mathbf{r}_0^T\mathbf{r}_0$
7: $k \leftarrow 1$
8: **while** $(\Delta > \epsilon)$ .AND. $(k < k_{\text{max}})$ **do**
9: $\mathbf{v}_{ap} \leftarrow \mathbf{mvm}(\mathbf{a}, \mathbf{p})$
10: $\alpha_k \leftarrow rTr_{\text{old}}/\mathbf{p}_k^T\mathbf{v}_{ap}$
11: $\mathbf{x}_k \leftarrow \mathbf{x}_{k-1} + \alpha_k\mathbf{p}_k$
12: $\mathbf{r}_k \leftarrow \mathbf{r}_{k-1} - \alpha_k\mathbf{v}_{ap}$
13: $rTr_{\text{new}} \leftarrow \mathbf{r}_k^T\mathbf{r}_k$
14: $\beta_k \leftarrow rTr_{\text{new}}/rTr_{\text{old}}$
15: $rTr_{\text{old}} \leftarrow rTr_{\text{new}}$
16: $\mathbf{p}_{k+1} \leftarrow \mathbf{r}_k + \beta_k\mathbf{p}_k$
17: $\Delta \leftarrow \|\mathbf{r}_k\| \cdot {}_{\text{over}}b_{\text{norm}}$
18: k_{++}
19: **end while**
20: **return** $(\mathbf{x}_{k-1})$
21: **end algorithm**

Figure 3.5 tuned software version of CG

analysis of CG showed that the matrix-vector multiply, which is $O(n^2)$, was the most expensive part of the kernel. Therefore, CG was refactored such that matrix-vector

multiply was a separate routine. Subsequently, `gprof` was used to profile the software version of the code. Not surprisingly, matrix vector multiply consumed nearly 67 percent of the runtime,therefore, according to Amdahl's law, the fraction to be enhanced $f_e = 0.67$.

3.5.2 Hardware

The tuned hardware CG algorithm is exactly as previously discussed , except a call to MVM is a call to hardware, as shown in Figure 3.6. The objective of mapping algorithms onto HPHCs is to obtain a speedup relative to the performance on a GPP. Anticipated overall speedup can be quantified via Amdahl's Law (J. L. Hennessy and D. A. Patterson 2003)

$$s_o = \frac{1}{1 - f_e + f_e/s_e},$$

where s_o is the overall speedup, f_e is the fraction of the system to be enhanced, and s_e is the speedup of the portion to be enhanced.

The SRC-7 MAPstation used as the target platform has two Intel Xeon processors and two Altera FPGAs running at 3.0GHz and 150MHz, respectively. In the absence of parallelization or pipelining, the FPGA runs *20x* slower than the GPP, i.e.:

$$s_e = \frac{150Mhz}{3000Mhz} = \frac{1}{20}$$

```
1:  algorithm MvMhw(kval, kcol, kptr, v, p, *first)
2:      if (*first) then                                  // grab the A matrix
3:          parBegin                                      // only two GCM banks so
4:              BUF_DMA_GCM1:OBM (kval, stripe-8)
5:              BUF_DMA_GCM2:OBM (kcol, stripe-8)
6:          parEnd
7:          parBegin                                      // parallel DMA limited to 2
8:              STREAM_DMA_GCM1:BRAM (kptr)
9:              STREAM_DMA_GCM2:BRAM (p)
10:         parEnd
11:     else                                              // matrix A is already in memory
12:         STREAM_DMA_GCM2:BRAM (p)                      // still need p
13:     end if
14:     p1 ← ... ← p8 ← p
15:     parBegin                                          // compute v = Ap
          p1:                                             // feed values stream
16:       for i in [1, kn_z] do
17:           a ← (a1 ··· a8) stripe-8 from kval_i
18:           j ← (j1 ··· j8) stripe-8 from kcol_i
19:           p ← (p1_j1 ··· p8_j8)
20:           V_FIFO ← dot8Tree (a, p)
21:       end for
          p2:                                             // feed counts stream
22:       for i in [1, n] do
23:           C_FIFO ← kptr_i+1 − kptr_i
24:       end for
          p3:                                             // streaming accumulator
25:       S_FIFO ← ∑_STREAM(V_FIFO, C_FIFO)
          p4:                                             // the dotN's
26:       for i in [1, n] do
27:           v_i ← S_FIFO
28:       end for
29:     parEnd
30:     STREAM_DMA_BRAM:GCM2 (v)                          // all done, return results
31: end algorithm
```

Figure 3.6 tuned hardware version of CG

The width of the dot product tree is 8 and the depth of the pipeline is approximately 50 (recall that the speed-up of a pipeline for a large number of items is the same as the depth). Thus, parallization and pipeling results in

$$s_e = \frac{8 \times 50}{20} = 20.$$

There are some serial operations and latencies that cannot be hidden during computation, so a conserative value is $s_e \approx 10$. Therefore, according to Amdahl's Law, an overall speedup $s_o = 1/(0.33 + 0.67/10) = 2.5$ is anticipated.

CHAPTER 4

RESULTS, ANALYSIS, AND COMPARISON

4.1 Results

4.1.1 Monolithic results

As shown in table 4.1, a monolithic approach might not produce a speedup. In our example, it produced a slowdown as depicted by the average of six samples of matrices. This is to be expected since the monolithic version of CG disregarded the rules and heuristics when mapping algorithms onto HPHCs.

Table 4.1 Slowdown using monolithic method

Matrix Name	Size (NZ)	t_{sw} (μs)	t_{hw} (μs)	Slowdown $\left(\frac{t_{\text{hw}}}{t_{\text{sw}}}\right)$
fass1.128.mtx	438	66381	258797	3.89
fass2.500.mtx	2298	55247	435985	7.89
fass1.1024.mtx	4918	396983	286116	0.72
fass1.2048.mtx	10038	74304	300623	4
fass4.4096.mtx	26270	398214	3759980	9.4
fass1.8192.mtx	40758	150452	490218	3.25
			Avg. slowdown	4.85

4.1.2 Tuned results

In contrast, following the heuristics, discussed in Section 3.2, it is possible to achieve a speedup. Due to technical difficulties, our target platform is undergoing repairs at the vendor's site, therefore, it is not possible to show the results of the refactored CG hardware mapping. Nonetheless, previous research mapping scientific kernels such as Transactions on Parallel and Distributed Systems' (TPDS) paper "Mapping a Jacobi Iterative Solver onto a High Performance Heterogeneous Computer" resulted in a 3-fold increase in performance as illustrated in Fig 4.1.

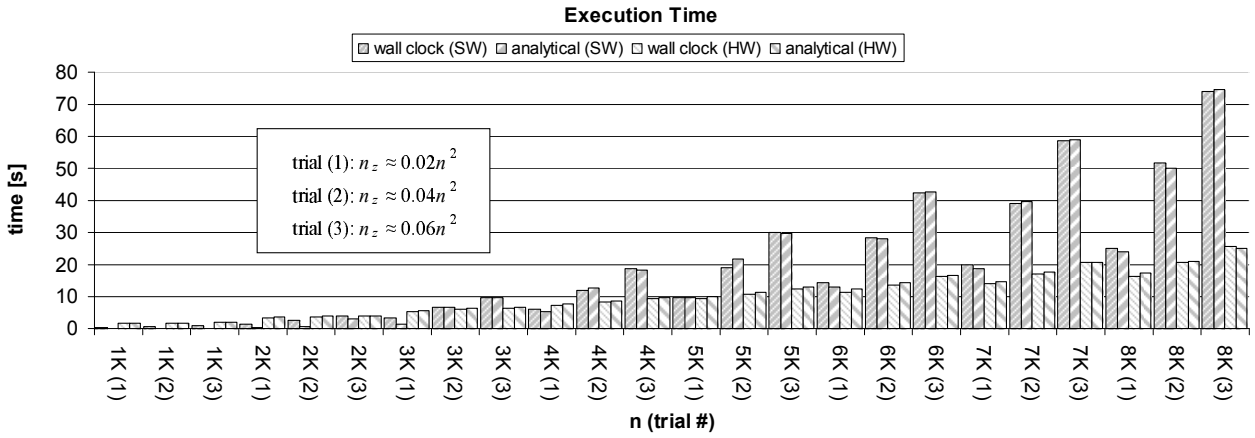

Figure 4.1 Run time comparison

The refactored CG algorithm that calls the hardware MVM is in close agreement with these speedups based upon the static analysis in Section 3.5.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

The research thus far has suggested strong evidence of an overall speed up of conjugate gradient (CG) when the rules and heuristics developed by the Jackson State University's High Performance Heterogeneous Computers (HPHC) research group are adhered to. The negative impact of performance has also been illustrated when these rules and heuristics are ignored (Hawkins, Alfarra, Morris, and Abed). Finding the "sweet spot" when mapping scientific kernels onto HPHCs has indeed helped in solidifying the veracity of the University's extensive research within the HPHC research domain (Rice, Abed, and Morris 2009; Rice, Pace, Gates, Morris, and Abed 2008; Justin L. Rice and Morris 2009; Morris, Silas, and Abed 2012; Morris 2006). Due to our target platform being upgraded from ARO funds at this current time, further progress in the research had to be postponed until the HPHC returns from the vendor. Future work includes finishing the hardware implementation of matrix vector multiplication (MVM) and calculating the actual overall speedup of CG.

CHAPTER 6

APPENDIX

APPENDIX A

APPLICATION DEVELOPMENT FILES

A.1 Monolithic Conjugate Gradient Development

A.1.1 main.c

```
1  /* main.c
2    sparse matrix jacobi iterative solver for SRC-7 hardware
3    Gerald R. Morris, gerald.r.morris@us.army.mil, 1 Mar 10
4    refactored by Gerald R. Morris, 7 Apr 2011 to use a
5    residual norm-based convergence test
6    Refactored by Jamory D. Hawkins and Anas Alfarra,Dec 2013
7    to use conjugate gradient as the iteration method
8
9    There are actually two options when building this code
10   The options are selected via defines in the cg.h header file
11   #define SWVER    or   #define HWVER
12   The first is the software version which employs the standard
13   nested loop approach to do the calculation serially.
14   The second is the k-aligned hardware version, which employ loops
15   wherein the matrix is dealt with in chunks of k elements at a time.
16   k-alignment requires an exact multiple of k elements in a row
17   The k-alignment process simply pads the matrix with 0's
18 */
19 #include <stdio.h>
20 #include <stdlib.h>
21 #include <sys/time.h>
22 #include <string.h>
```

```
23 #include <stdint.h>
24 #include <math.h>
25 #include "cg.h"
26 #include "mmio.h"
27 #include "sparsekit.h"
28 #include "usec.h"
29 #ifdef HWVER
30     #include <libmap.h>
31 #endif
32
33 // define this when we want a verbose mode (mostly debug stuff)
34 //#define VURBOZE
35
36 int main (int argc, char *argv[]) {
37
38     // matrices are stored in MatrixMarket coordinate format
39     // we use the MatrixMarket mmio (I/O) routines
40     char fname[255];          // MatrixMarket file
41     FILE *fmm;                // MatrixMarket file pointer
42     MM_typecode matcode;      // MatrixMarket meta-data
43     int n;                    // # matrix rows
44     int nc;                   // # matrix columns
45     int nnz;                  // # non-zero elements
46     float *acoo;              // matrix values
47     int *icoo;                // matrix row indices
48     int *jcoo;                // matrix column indices
49
50     // cg uses compressed sparse row format
51     float *val;               // CSR sparse matrix (input matrix)
52     int *col;                 // CSR column index
53     int *ptr;                 // CSR row pointer
54
```

```
55      float *x;                   // x vector
56      float *b;                   // constant vector
57      float b2n_1;                // (2-norm of b vector)^(-1)
58      float r2nb2n;               // ratio of 2-norms (convergence test)
59
60      char result[1024];          // capture results
61      int iters;                  // number of iterations needed
62      FILE *fres;                 // results file pointer
63
64      double t0,t1,t2,t3;         // to capture wall clock run time
65      int i,j,k;                  // loop indices
66
67  #ifdef HWVER  // if it's the hardware version, we need these
68      // for k-aligned CSR data
69      float *kval;                // CSR matrix k-aligned values
70      int *kcol;                  // CSR k-aligned column index
71      int *kptr;                  // CSR k-aligned row pointer
72      int knz;                    // k-aligned # of non-zeros
73      int kvalsz;                 // allocation size of kval (bytes)
74      int kcolsz;                 // allocation size of kcol
75      int kptrsz;                 // allocation size of kptr
76
77      float *D_1;                 // D^(-1) (1/aii as a vector)
78      int D_1sz,bsz,xsz;          // allocation size of vectors
79
80      // we'll allocate segments using both OBCM banks
81      // and marshall the MAP data in those banks (speed)
82      gcm_seg_desc_t *sd1;        // OBCM segment descriptor pointers
83      gcm_seg_desc_t *sd2;        // for bank 1 and bank 2
84      uint64_t sd1sz;             // segment sizes
85      uint64_t sd2sz;
86      // address of the MAP data allocated from the 2 segments above
```

```
87      gcm_addr_t kvalA;          // OBCM address of CSR matrix values
88      gcm_addr_t kcolA;          // OBCM address of CSR column indices
89      gcm_addr_t kptrA;          // OBCM address of CSR row pointers
90      gcm_addr_t bA;             // OBCM address of constant vector
91      gcm_addr_t D_1A;           // OBCM address of 1/aii n-vector
92      gcm_addr_t xA;             // OBCM address of approximate solution
93
94      // for MAP allocation and usage
95      int nmaps = 1;             // the number of MAPs needed
96      int mapno = 0;             // first MAP number is 0
97  #endif // #ifdef HWVER
98
99  #ifdef VURBOZE  // format strings, etc.. when we run in verbose mode
100     char *f1="\n%5d:    %9.2e     %9.2e     %9.2e     %9.2e\n";
101     char *f2="          (%4d,%4d)  (%4d,%4d)  (%4d,%4d)  (%4d,%4d)";
102     char *f3="\n%5d:%9.2e%9.2e%9.2e%9.2e%6d%5d%5d%5d";
103     char *r1="               val                            col";
104     char *k1="               kval                          kcol";
105     char *r2=" 0         1         2         3         0    1    2    3";
106     char *r3="-------------------------------------------------------------------";
107     char *indent = "           ";
108     char *spc="...";
109     int head = 16;             // 1st few matrix values
110 #endif // #ifdef VURBOZE
111
112     t0 = usec();               // first start time
113
114     // matrix file name is specified on the command line
115     if (argc == 2) {
116         strcpy(fname,argv[1]);
117     } else {
118         fprintf(stderr,"*** Usage: ./cg MM_file\n");
```

```
119         exit (1);
120     }
121
122     // open the input and result files
123     if ((fmm = fopen(fname, "r")) == NULL) {
124         fprintf(stderr, "Cannot open file %s\n", fname);
125         exit(1);
126     }
127     fprintf(stderr,"Reading %s\n",fname);
128     if ((fres = fopen ("result", "w")) == NULL) {
129         fprintf (stderr, "failed to open file 'result'\n");
130         exit (1);
131     }
132
133     // get matrix information from MatrixMarket file
134     mm_read_banner(fmm, &matcode);
135     mm_read_mtx_crd_size(fmm, &n, &nc, &nnz);
136     fprintf(stderr,"   %dx%d %s with %d nonzero values\n",
137       n,nc,mm_typecode_to_str(matcode),nnz);
138
139     // allocate coordinate format arrays now that sizes are known
140     acoo = (float *) malloc(nnz * sizeof(float));
141     icoo = (int *) malloc(nnz * sizeof(int));
142     jcoo = (int *) malloc(nnz * sizeof(int));
143
144     // read MatrixMarket coordinate-format matrix data
145     for (i=0; i<nnz; i++) {
146         fscanf(fmm, "%d %d %g\n", &icoo[i], &jcoo[i], &acoo[i]);
147         icoo[i]--;  // adjust from 1-based to 0-based for C arrays
148         jcoo[i]--;
149     }
150     fclose(fmm);
```

```
151
152 #ifdef VURBOZE // dump up to head elements of input matrix
153     printf("%s%s","COO:       ",r3);
154     for (i=0; i<MIN(nnz,3*head); i+=4) {
155         printf(f1,
156           i,acoo[i],acoo[i+1],acoo[i+2],acoo[i+3]);
157         printf(f2,icoo[i],jcoo[i],icoo[i+1],jcoo[i+1],
158           icoo[i+2],jcoo[i+2],icoo[i+3],jcoo[i+3]);
159     }
160     printf("    %s\n",spc);
161     printf("%s%s\n",indent,r3);
162 #endif // #ifdef VURBOZE
163
164     // allocate CSR array and convert to CSR format
165     val = (float *) malloc(nnz * sizeof(float));
166     col = (int *) malloc(nnz * sizeof(int));
167     ptr = (int *) malloc((n+1) * sizeof(int));
168     coocsr(n,nnz,acoo,icoo,jcoo,val,col,ptr);  // SPARSKIT routine
169
170     // have CSR data so deallocate MatrixMarket arrays
171     free(acoo);
172     free(icoo);
173     free(jcoo);
174
175 #ifdef VURBOZE  // show the CSR format matrix data
176     // val, and col vectors
177     printf("%s%s\n",indent,r1);
178     printf("%s%s\n",indent,r2);
179     printf("%s%s","CSR:       ",r3);
180     for (i=0; i<MIN(nnz,3*head); i+=4) {
181         printf(f3,
182           i,val[i],val[i+1],val[i+2],val[i+3],
```

```
183                 col[i],col[i+1],col[i+2],col[i+3]);
184         }
185         printf("    %s\n",spc);
186
187         // and ptr array  (sideways)
188         printf("%s%s\n",indent,r3);
189         printf("   i:     ");
190         for (i=0; i<=MIN(n,head); i++) {
191             printf("%5d",i);
192         }
193         printf("  %s\n",spc);
194         printf("   ptr:   ");
195         for (i=0; i<=MIN(n,head); i++) {
196             printf("%5d",ptr[i]);
197         }
198         printf("  %s\n",spc);
199         printf("%s%s\n",indent,r3);
200 #endif // #ifdef VURBOZE
201
202 #ifndef HWVER
203         // SWVER: allocate solution and constant vector
204         x = (float *) malloc (n * sizeof(float));
205         b = (float *) malloc (n * sizeof(float));
206
207 #else
208         // HWVER: allocate and load the matrix into cache-aligned buffers.
209         // as part of software/hardware codesign tradeoffs the
210         // k-alignment of A will be done here in software
211         // the MAP wants the CSR vectors to be aligned on cache
212         // boundaries.  We can't place this restriction on upper-level
213         // callers, so we will marshall the data here into
214         // properly aligned arrays.  Since we have to do the memcopy
```

```
215     // anyway, we'll just go ahead and do the k-alignment needed
216     // by our MAP routine here in software.
217
218     // figure out the growth in kval and kcol due to the
219     // k-alignment processing, i.e., calculate knz
220     knz = 0;
221     for (i=0; i<n; i++) {           // loop across all matrix rows
222         for (j=ptr[i]; j<ptr[i+1]; j++) {
223             knz++;                  // count # non-zeros
224         }
225         while ((knz%K) != 0) {      // pad with 0's if necessary
226             knz++;                  // to make knz a multiple of K
227         }
228     }
229
230     // now we have the knz value, so we can calculate array sizes
231     // use SALIGN macro (cg.h) to avoid cache boundary problems
232     kvalsz = SALIGN(knz*sizeof(float),CACHEALIGN);
233     kcolsz = SALIGN(knz*sizeof(int),CACHEALIGN);
234     kptrsz = SALIGN((n+1)*sizeof(int),CACHEALIGN);
235     D_1sz = SALIGN(n*sizeof(float),CACHEALIGN);
236     xsz = SALIGN(n*sizeof(float),CACHEALIGN);
237     bsz = SALIGN(n*sizeof(float),CACHEALIGN);
238
239     // allocate arrays for the k-aligned CSR matrix
240     kval = (float*)Cache_Aligned_Allocate (kvalsz);
241     kcol = (int*)  Cache_Aligned_Allocate (kcolsz);
242     kptr = (int*)  Cache_Aligned_Allocate (kptrsz);
243     D_1 = (float*) Cache_Aligned_Allocate (D_1sz);
244
245     // also need OBCM bank segment descriptors
246     // to speed up MAP processing, place data in separate banks
```

```
247     // kval, kptr, and D_1 go into OBCM bank 1
248     // kcol, b, and x go into OBCM bank 2
249     // per SRC7CPEGuide (pg 57), must start on 8-byte
250     // boundaries and be allocated in length that is a multiple of 8
251     // this is already guaranteed because of our cache aligned sizes
252     sd1sz = (uint64_t)kvalsz + (uint64_t)kptrsz + (uint64_t)D_1sz;
253     sd2sz = (uint64_t)kcolsz + (uint64_t)bsz + (uint64_t)xsz;
254     if (gcm_allocate_seg_by_bank(sd1sz, 1, &sd1)) {
255         fprintf(stderr,"Bank 1 OBCM segment allocation failed\n");
256         exit(1);
257     }
258     if (gcm_allocate_seg_by_bank(sd2sz, 2, &sd2)) {
259         fprintf(stderr,"Bank 2 OBCM segment allocation failed\n");
260         exit(1);
261     }
262
263     // now allocate the buffer addresses in OBCM segments
264     if (!(kvalA = gcm_allocate_from_seg(kvalsz,sd1))) {
265         fprintf(stderr,"OBCM allocation for kval failed\n");
266         exit(1);
267     }
268     if (!(kcolA = gcm_allocate_from_seg(kcolsz,sd2))) {
269         fprintf(stderr,"OBCM allocation for kcol failed\n");
270         exit(1);
271     }
272     if (!(kptrA = gcm_allocate_from_seg(kptrsz,sd1))) {
273         fprintf(stderr,"OBCM allocation for kptr failed\n");
274         exit(1);
275     }
276     if (!(bA = gcm_allocate_from_seg(bsz,sd2))) {
277         fprintf(stderr,"OBCM allocation for b failed\n");
278         exit(1);
```

```
279     }
280     if (!(D_1A = gcm_allocate_from_seg(D_1sz,sd1))) {
281         fprintf(stderr,"OBCM allocation for D_1 failed\n");
282         exit(1);
283     }
284     if (!(xA = gcm_allocate_from_seg(xsz,sd2))) {
285         fprintf(stderr,"OBCM allocation for x failed\n");
286         exit(1);
287     }
288
289     // k-align the kval, kcol, and kptr arrays and compute D_1
290     // the meaning of kptr is slightly different than ptr
291     // recall, MAP will be grabbing kcol and kval elements
292     // K per clock cycle, so kptr expresses things in groups of K
293     knz = 0;
294     for (i=0; i<n; i++) {                   // all matrix rows
295         kptr[i] = knz/K;                    // k-aligned ptr value
296         for (j=ptr[i]; j<ptr[i+1]; j++) {
297             kval[knz] = val[j];
298             if (i==col[j]) {                // aii (diagonal value)
299                 D_1[i]=1/val[j];            // calculate 1/aii
300             }
301             kcol[knz++] = col[j];
302         }
303         while ((knz%K) != 0) {              // pad with 0's
304             kval[knz] = 0.0;                // to fill a group of K
305             kcol[knz++] = 0;
306         }
307     }
308     kptr[i] = knz/K;                        // similar to ptr[n+1]
309
310 #ifdef VURBOZE  // show the k-aligned sizes and data
```

```
311     printf("words:  n,knz = %d,%d\n",n,knz);
312     printf("\nbytes:  |kval|=%d,|kcol|=%d,|kptr|=%d,|vec|=%d\n",
313       kvalsz,kcolsz,kptrsz,xsz);
314
315     // show the k-aligned kval, and kcol vectors
316     printf("\n%s%s\n",indent,k1);
317     printf("%s%s\n",indent,r2);
318     printf("%s%s","kCSR:      ",r3);
319     for (i=0; i<MIN(knz,3*head); i+=4) {
320         printf(f3,
321           i,kval[i],kval[i+1],kval[i+2],kval[i+3],
322             kcol[i],kcol[i+1],kcol[i+2],kcol[i+3]);
323     }
324     printf("   %s\n",spc);
325
326     // show the k-aligned kptr array (sideways)
327     printf("\n   i:     ");
328     for (i=0; i<=MIN(n,head); i++) {
329         printf("%4d",i);
330     }
331     printf("  %s\n",spc);
332     printf("   kptr:  ");
333     for (i=0; i<=MIN(n,head); i++) {
334         printf("%4d",kptr[i]);
335     }
336     printf("  %s\n",spc);
337     printf("   D_1:  ");
338     for (i=0; i<=MIN(n,head); i++) {
339         printf("%6.2e",D_1[i]);
340     }
341     printf("  %s\n",spc);
342     printf("%s%s\n",indent,r3);
```

```
343 #endif // #ifdef VURBOZE
344
345     // allocate solution and constant vector
346     x = (float*)Cache_Aligned_Allocate (xsz);
347     b = (float*)Cache_Aligned_Allocate (bsz);
348
349     if (map_allocate(nmaps)) {  // allocate the MAP
350         fprintf(stderr,"Could not allocate MAP.  Bye!\n");
351         exit(1);
352     }
353
354 #endif // HWVER
355
356     // calc constant vector b assuming xcur[i] = 1000.0
357     // we'll dump the b vector to our result file
358     // also calculate 2-norm of b vector
359     b2n_1 = 0.0;
360     for (i=0; i<n; i++) {
361         b[i] = 0.0;
362         for (j=ptr[i]; j<ptr[i+1]; j++) {   // sum(aij*xj)
363             b[i]+=val[j]*1000.0;
364         }
365         b2n_1 += b[i]*b[i];                    // sum(b[i]^2)
366         fprintf(fres,"b[%d] = %12.10e\n",i,b[i]);
367     }
368     b2n_1 = 1/sqrt(b2n_1);          // (2-norm of b vector)^(-1)
369
370     // all of that just to get our matrix and constant vectors!
371     // now we do the software or hardware version of cg
372
373 #ifndef HWVER
374     // SWVER: do the cg 5 times to get good average
```

```
375     t1 = usec();            // first stop time, second start time
376     for(k=0;k<5;k++) { // run it 5 times to get a good average
377         // initialize the x vector to all zeros
378         // basically we start "guessing" at x[i] = 0
379         for (i=0; i<n; i++) {
380             x[i] = 0.0;
381         }
382         cg(val,col,ptr,b,n,nnz,x,b2n_1,&r2nb2n,&iters);  // solve for x
383     }
384     t2 = usec();            // second stop time, third start time
385
386 #else
387     // HWVER: do the cg 5 times to get good average
388     t1 = usec();            // first stop time, second start time
389     for(k=0;k<5;k++) { // run it 5 times to get a good average
390         // copy data to the OBCM buffers
391         gcm_cp_to(kvalA, kval, kvalsz);
392         gcm_cp_to(kcolA, kcol, kcolsz);
393         gcm_cp_to(kptrA, kptr, kptrsz);
394         gcm_cp_to(bA, b, bsz);
395         r2nb2n = EPSILON + 1;    // assume not yet converged
396         cg(kvalA,kcolA,kptrA,bA,n,knz,xA,b2n_1,&r2nb2n,&iters,mapno);
397         gcm_cp_from(xA, x, xsz); // copy result back from OBCM buffer
398     }
399     t2 = usec();            // second stop time, third start time
400
401 #endif // HWVER
402
403     // save results
404     for (i=0; i<n; i++) {
405         fprintf(fres, "x[%d] = %12.10e\n",i,x[i]);
406     }
```

```
407
408     // close output file and release memory
409     fclose(fres);
410
411     free(val);
412     free(col);
413     free(ptr);
414
415     // release all the stuff that was allocated
416 #ifndef HWVER
417     free(b);
418     free(x);
419 #else
420     // deallocate the MAP
421     if (map_free(nmaps)) {
422         fprintf(stderr,"Could not deallocate MAP.  Bye!\n");
423         exit(1);
424     }
425     // arrays
426     Cache_Aligned_Free((char*)kval);
427     Cache_Aligned_Free((char*)kcol);
428     Cache_Aligned_Free((char*)kptr);
429     Cache_Aligned_Free((char*)D_1);
430     Cache_Aligned_Free((char*)b);
431     Cache_Aligned_Free((char*)x);
432     // OBCM buffers
433     gcm_free(kvalA);
434     gcm_free(kcolA);
435     gcm_free(kptrA);
436     gcm_free(D_1A);
437     gcm_free(bA);
438     gcm_free(xA);
```

```
439     // OBCM segments
440     gcm_free_seg(sd1);
441     gcm_free_seg(sd2);
442 #endif // HWVER
443
444     // report to stdout and quit
445     t3 = usec();              // third stop time
446     sprintf(result,"%s \t %10d \t %6d \t %12.10f \t %12.10f",
447       fname,nnz,iters,r2nb2n,
448       (t1 - t0) + (t2 - t1)/5 + (t3 - t2));
449     puts(result);
450     if(iters<MAXITERS) {
451         exit(0);
452     } else {
453         sprintf(result,"*** Maximum iterations (%d) exceeded...",MAXITERS);
454         puts(result);
455         exit(1);
456     }
457 }
```

A.1.2 cg.h

```
458 /* cg.h
459    J. Hawkins, A. Alfarra, and G. Morris, Dec 2013
460    prototype for sparse matrix conjugate gradient
461 */
462
463 #include <stdio.h>
464 #include <stdlib.h>
465
466 #define HWVER
467 #ifdef HWVER
468     #include <libmap.h>
```

```
469 #endif
470
471 // sometimes MIN and MAX macros already defined
472 #ifndef MIN
473     #define MIN(x,y) (((x)<(y)?(x):(y)))
474     #define MAX(x,y) (((x)>(y)?(x):(y)))
475 #endif
476
477 // width of the binary tree, maximum n,
478 // maximum iterations, convergence criterion
479 #define K (8)
480 #define NMAX (8192)
481 #define MAXITERS (100000)
482 #define EPSILON (1e-6)
483
484 #ifndef HWVER
485
486     // SWVER: the software cg iteration function
487     void cg(
488       float val[],   // CSR sparse matrix values
489       int col[],     // CSR column indices
490       int ptr[],     // CSR row pointers
491       float b[],     // known constant vector
492       int n,         // matrix order
493       int nnz,       // number of non-zeros
494       float x[],     // solve for x
495       float b2n_1,   // (2-norm of b vector)^(-1)
496       float *r2nb2n, // ratio of 2-norms (convergence test)
497       int *iters     // iterations to solve or give up
498     );
499
500 #else
```

```
501
502     // HWVER: the hardware cg iteration function
503     void cg(
504       gcm_addr_t kvalA, // GCM address of CSR matrix values
505       gcm_addr_t kcolA, // GCM address of CSR column indices
506       gcm_addr_t kptrA, // GCM address of CSR row pointers
507       gcm_addr_t bA,    // GCM address of constant vector
508       int n,            // number of matrix rows and columns
509       int knz,          // k-aligned number of non-zeros
510       gcm_addr_t xA,    // GCM address of approximate solution
511       float b2n_1,      // (2-norm of b vector)^(-1)
512       float *r2nb2n,    // ratio of 2-norms (convergence test)
513       int *iters,       // iterations needed to converge (or give up)
514       int mapnum);      // MAP number required by Carte
515
516     // cache alignment (% more /proc/cpuinfo)
517     #define CACHEALIGN (128)
518
519     // return next integer multiple of sz >= x
520     #define SALIGN(x,sz) ((((x)%(sz))==0)?((x)):((x)+(sz)-((x)%(sz))))
521
522 #endif // HWVER
```

A.1.3 cg.c

```
523 /* cg.c
524    J. Hawkins, A. Alfarra, and G. Morris, Dec 2013
525    software version of sparse matrix conjugate gradient
526 */
527 #include <stdlib.h>
528 #include <math.h>
529 #include "cg.h"
530
```

```
531 void cg(
532   float val[],    // CSR sparse matrix values
533   int col[],      // CSR column index
534   int ptr[],      // CSR row pointer
535   float b[],      // known constant vector
536   int n,          // matrix order
537   int nnz,        // number of non-zeros
538   float x[],      // solve for x
539   float b2n_1,    // (2-norm of b vector)^(-1)
540   float *r2nb2n, // ratio of 2-norms (convergence test)
541   int *iters) {  // iterations to solve or give up
542
543     /* x_0 = 0 so Ax_0 = 0 , ergo vector Ax not needed
544     float *Ax;               // for initial search direction */
545     float *p;                // search direction
546     float *r;                // residual vector
547     float *v_ap;             // avoid multiple Ap mvm
548     float alpha;             // step size
549     float rTrold;            // dot(r(k-1),r(k-1))
550     float rTrnew;            // dot(r(k),r(k))
551     float beta;              // projection operator
552     float normr;             // 2-norm of r (||r||)
553     float delta;             // residual norm = ||r||/||b||
554     int iterslcl = 1;        // local iteration index
555     float sum;               // multi-use accumulator
556     int i,j;                 // index variables
557
558     // allocate vectors
559     /* x_0 = 0 so Ax_0 = 0, ergo no vector needed
560     Ax = (float *)malloc(n*sizeof(float)); */
561     p = (float *)malloc(n*sizeof(float));
562     r = (float *)malloc(n*sizeof(float));
```

```
563     v_ap = (float *)malloc(n*sizeof(float));
564
565     // calculate Ax_0
566     /* x_0 = 0 so Ax_0 = 0, ergo no calculation
567     for(i=0;i<n;i++) {
568         sum = 0; // recall A is sparse
569         for(j=ptr[i];j<ptr[i+1];j++) {
570             sum += val[j]*x[col[j]];
571         }
572         Ax[i]=sum;
573     }
574     */
575
576     // algorithms lines 2,3,5,and 7
577     // recall Ax_0=0 so p1=r0=b
578     // 1st A-orthogonal search direction (p1)
579     // 0th residual (r0 = p1)
580     // rTrold = dot(r0,r0)
581     rTrold = 0;
582     for(i=0;i<n;i++) {
583         x[i] = 0;             // want x_0 = 0
584         p[i] = b[i];          // p1 = b-Ax_0
585         r[i] = p[i];          // r0 = p1
586         rTrold += r[i]*r[i]; // dot(r(0),r(0))
587     }
588
589     delta = EPSILON + 1.0;  // force loop entry
590     // loop until converge or max iters exceeded
591     while((delta>EPSILON)&&(iterslcl<MAXITERS)) {
592
593         // alg line 10: need Ap twice, so calc once
594         for(i=0;i<n;i++) {
```

```
595             sum = 0; // recall A is sparse
596             for(j=ptr[i];j<ptr[i+1];j++) {
597                 sum += val[j]*p[col[j]];
598             }
599             v_ap[i] = sum;
600         }
601
602         // alg 11: calc step size (alpha)
603         sum = 0;
604         for(i=0;i<n;i++) {
605             sum += p[i]*v_ap[i];
606         }
607         alpha = rTrold/sum;
608
609         // alg 12: calc next x and
610         // alg 16: residual vector and
611         // alg 18: dot (r,r)
612         rTrnew = 0;
613         for(i=0;i<n;i++) {
614             x[i] += alpha*p[i];
615             r[i] -= alpha*v_ap[i];
616             rTrnew += r[i]*r[i];
617         }
618         normr = sqrt(rTrnew); // ||r||, part of alg 22
619
620         beta = rTrnew/rTrold; // alg 19: projection operator
621
622         rTrold = rTrnew;      // alg 20: for next iteration
623
624         // alg 21: calc next search direction
625         for(i=0;i<n;i++) {
626             p[i] = r[i]+beta*p[i];
```

```
627           }
628
629           delta = normr * b2n_1;      // alg 22: residual norm
630           iterslcl++;                 // next iteration
631
632       } // end while
633
634       *iters = iterslcl;              // report iterations
635       *r2nb2n = delta;                // and residual norm
636
637       // release vectors
638       /* x_0 = 0 so Ax_0 = 0, ergo no Ax vector
639       free(Ax); */
640       free(p);
641       free(r);
642       free(v_ap);
643
644   } // end cg
```

A.1.4 cg.mc

```
645   // cg.mc
646   // sparse matrix CG iterative solver, Jerry Morris & Jamory Hawkins, Jan 2014
647   // partly based on Morris' April 2011 Jacobi solver
648   // this is the hardware version that runs on the SRC-7 MAP
649   //
650   // build:  make debug (debug version : ./cghw.dbg)
651   //         make hw (hardware version : ./cghw)
652   //
653   // algorithm cghw(kval,kcol,kptr,b,b2,n,knz,x)
654   //     // kval, kcol, kptr: CSR-format A matrix
655   //     // b: constant vector, b2 = 1/||b||
656   //     // n: matrix row & cols, knz: # non-zeros
```

```
657 //      // x: desired solution, i.e., Ax=b
658 //
659 //      // place data into OBM and BRAM
660 //      // only have two GCM banks so
661 //      // parallel DMA's limited to 2
662 //      parBegin
663 //          BUF_DMAGCM1:OBM (kval, stripe-8)
664 //          BUF_DMAGCM2:OBM (kcol, stripe-8)
665 //      parEnd
666 //      parBegin
667 //          STREAM_DMAGCM1:BRAM (kptr)
668 //          STREAM_DMAGCM2:BRAM (b)
669 //      parEnd
670 //
671 //      // set up for repeat loop
672 //      for i in [1,n] do // x0=0, so p1:=r0:=b
673 //          x[i] := 0
674 //          p[i] := r[i] := b[i]
675 //          MAC(r[i],r[i],rTrold) // calculate dot(r0,r0)
676 //      end for
677 //
678 //      d:=0    // loop index
679 //
680 //      repeat
681 //
682 //          // 11 copies of p vector
683 //          // to avoid loop-carried dependence
684 //          // 8 for dot-product tree
685 //          // 3 elsewhere
686 //          p1:=p2:= ... :=p11:=p
687 //
688 //          // compute v := Ap via streaming accumulator
```

```
689 //          parBegin
690 //
691 //              // dot8s into Vfifo stream
692 //              p1: for i in [1,knz] do
693 //                  a8 := (a1 ... a8) stripe-8 from kval[i]
694 //                  j8 := (j1 ... j8) stripe-8 from kcol[i]
695 //                  p8 :=(p1[j1]  p8[j8])
696 //                  Vfifo := dot(a8,p8)
697 //              end for
698 //
699 //              // # dot8s per row into Cfifo stream
700 //              p2: for i in [1,n] do
701 //                  Cfifo := kptr[i+1] - kptr[i]
702 //              end for
703 //
704 //              // n dot products into Sfifo
705 //              p3: Sfifo := Stream_Accum(Vfifo,Cfifo)
706 //
707 //              // dot[i]=dot(A[i],p)
708 //              p4: for i in [1,n] do
709 //                  v[i] := Sfifo
710 //                  MAC(v[i],p9[i],pTv) // dot(p,v)
711 //              end for
712 //
713 //          parEnd
714 //
715 //          alpha:=rTrold/pTv //stepsize
716 //
717 //          for i in [1,n] do
718 //              x[i] := x[i] + alpha*p10[i] // next point
719 //              r[i] := r[i]  alpha*v[i]    // residual
720 //              MAC(r[i],r[i],rTrnew)       // new dot(r,r)
```

```
721 //          end for
722 //
723 //          beta := rTrnew/rTrold           // projection operator
724 //
725 //          rTrold := rTrnew // for next iteration
726 //
727 //          for i in [1, n] do // new search direction
728 //              p[i] := r[i] + beta*p11[i]
729 //          end for
730 //
731 //          // residual norm
732 //          r2b2:= sqrt(rTrold)*b2 // ||r||= sqrt(rTrold)
733 //
734 //          d:=d+1 // next iteration
735 //
736 //      until (r2b2 <= epsilon .OR. d > dmax)
737 //
738 //      // all done, stream x back to GCM
739 //      STREAM_DMABRAM:GCM2 (x)
740 //
741 // end algorithm
742 //
743 // Derivation of residual norm termination condition
744 //
745 // Recall, the 2-norm of n-vector (say) y is given by
746 // ||y|| = sqrt(sum(i=1..n)(y[i]^2))
747 //
748 // rcur = A(xact - xcur) ->
749 // ||xact - xcur|| = ||A^(-1)*rcur|| <= ||A^(-1)||*||rcur|| [5]
750 //
751 // b = A*xact -> ||b|| = ||A*xact|| <= ||A||*||xact|| ->
752 // 1/||xact|| <= ||A||/||b|| [6]
```

```
753 //
754 // multiply [5] and [6]
755 // ||xact - xcur||/||xact|| <= ||A||*||A^(-1)||*(||rcur||/||b||)
756 //
757 // condition number, k(A) = ||A||*||A^(-1)|| = lambda_max/lambda_min
758 // where the lambda are max and min eigenvalues of matrix A.
759 // In practice, finding the eigenvalues is at least as hard as solving
760 // the set of equations, furthermore 1<=k(A)<infinity,
761 // so we can simply use the ratio of 2-norms
762 // as our termination, i.e., we want
763 //
764 // ||rcur||/||b|| < epsilon
765 //
766 // This is the k-aligned hardware version.  There is an option for
767 // a pure software version of cg.  The versions are selected
768 // via a variable that is found in the cg.h header file
769 //
770 #include <libmap.h>
771 #include "cg.h"
772 void cg(
773   gcm_addr_t kvalA, // GCM address of CSR matrix values
774   gcm_addr_t kcolA, // GCM address of CSR column indices
775   gcm_addr_t kptrA, // GCM address of CSR row pointers
776   gcm_addr_t bA,    // GCM address of constant vector
777   int n,            // number of matrix rows and columns
778   int knz,          // k-aligned number of non-zeros
779   gcm_addr_t xA,    // GCM address of approximate solution
780   float b2n_1,      // (2-norm of b vector)^(-1)
781   float *r2nb2n,    // ratio of 2-norms (convergence test)
782   int *iters,       // iterations needed to converge (or give up)
783   int mapnum) {     // MAP number required by Carte
784
```

```
785     // k-striped local copy of k-aligned matrix values (kval) in OBM
786     // 2-pack (0,1)
787     // 2-pack (2,3)
788     // 2-pack (4,5)
789     // 2-pack (6,7)
790     OBM_BANK_A(kvalL01, int64_t, MAX_OBM_SIZE)
791     OBM_BANK_B(kvalL23, int64_t, MAX_OBM_SIZE)
792     OBM_BANK_C(kvalL45, int64_t, MAX_OBM_SIZE)
793     OBM_BANK_D(kvalL67, int64_t, MAX_OBM_SIZE)
794
795     // k-striped local copy of k-aligned matrix columns (kcol) in OBM
796     // 2-pack (0,1)
797     // 2-pack (2,3)
798     // 2-pack (4,5)
799     // 2-pack (6,7)
800     OBM_BANK_E(kcolL01, int64_t, MAX_OBM_SIZE)
801     OBM_BANK_F(kcolL23, int64_t, MAX_OBM_SIZE)
802     OBM_BANK_G(kcolL45, int64_t, MAX_OBM_SIZE)
803     OBM_BANK_H(kcolL67, int64_t, MAX_OBM_SIZE)
804
805     // all other vectors go into BRAM on the Altera FPGA
806     // we play some games to balance use of the BRAM memory.
807     // EP2S180 FPGA has 768 M4K RAM, 930 M512 RAM, and 9 MRAM blocks.
808     // we consume 662 of 768 M4K BRAMs, 81 of 930 M512 BRAMs,
809     // and 9 of 9 MRAM blocks.  Basically we declare some arrays
810     // bigger than they need to be so that Carte puts them in
811     // MRAM rather than M4K.  see SRC-7 Carte C Programming
812     // Environment Guide for additional details.
813
814     int kptrL[2*NMAX];    // local copies of kptr
815     float bL[NMAX];       // local copy of b
816     float r[NMAX];        // residual vector
```

```
817     float v[NMAX];          // matrix vector multiply: v = Ap
818     float alpha;            // step size
819     float beta;             // projection operator
820     float rTrold;           // previous residual dot(r,r)
821     float rTrnew;           // current residual dot(r,r)
822     float pTv;              // dot(p,v)
823     float ri;               // avoid mixed read/write
824
825     // the dot8 tree needs independent copies of p
826     // to avoid multiple-cycle memory accesses
827     float pC0[NMAX];        // the p copy (pC*) are the
828     float pC1[NMAX];        // p inputs for each leaf node
829     float pC2[NMAX];        // of a size k=8 dot product unit
830     float pC3[2*NMAX];      // the dot8 is fully pipelined
831     float pC4[2*NMAX];      // the array A inputs to the dot8
832     float pC5[2*NMAX];      // come from the striped OBM banks
833     float pC6[2*NMAX];
834     float pC7[2*NMAX];
835
836     float pC8[NMAX];        // we need 3 additional copies of p
837     float pC9[NMAX];        // for other calculations that
838     float pCa[NMAX];        // access the p vector
839
840     float pnxt[NMAX];       // we need this to calculate the next p
841     float xL[2*NMAX];       // local copy for the x calculations
842
843     // input vector stream processing is as follows:
844     // 1.  DMAstream in 264-bit packed values from GCM
845     // 2.  stream width convert into 32-bit stream
846     // 4.  unload single 32-bit stream into local BRAM arrays
847     Stream_256 Skptr256;              // kptr streams
848     Stream_32 Skptr32;
```

```
849     Stream_256 Sb256;                   // b streams
850     Stream_32 Sb32;
851
852     // the streaming accumulator bears explanation
853     //
854     // suppose I have 3 vectors: a=(1,2,3), b=(4,5), c=(6,7,8,9)
855     // and I want the three vector sums: sa=1+2+3=6, sb=9, sc=30
856     // further suppose the vector data is being delivered as
857     // a single stream, vals = 1,2,3,4,5,6,7,8,9
858     // and I have another stream, cnts = 3,2,4 (sizes of a, b, c)
859     // I use the streaming accumulator to produce
860     // a third stream, sums = 6,9,30.
861     // Here's an elided pseudo-code of the four parallel sections
862     // that one would use to accomplish this task
863     //
864     // pars {
865     // par { // stream out values
866     //     for i=1,9 {
867     //         put_stream(&Sva, vals[i])
868     //     }
869     // }
870     // par {  // stream out counts
871     //     for i=1,3 {
872     //         put_stream(&Sca,cnts[i]))
873     //     }
874     // par { // eat vals, cnts and stream out sums
875     //     stream_fp_accum_strm_counts_32_rr_term(&Sva,&Sca,&Ssa)
876     // }
877     // par { // save sums
878     //     for i=1,3 {
879     //         get_stream(&Ssa,&sums[i]);
880     // }
```

```
881     // }
882     //
883     // in our case accumulator stream processing is as follows:
884     // 1.  compute dot8 patial dot products: stream dot8 vals to accum
885     // 2.  compute number of dot8s per row: stream cnts to accum
886     // 3.  accum sums dot8s & outputs n full dot products (dotNs)
887     // 4.  complete processing
888     //
889     Stream_32 Sva;      // value stream (dot8s) to accum
890     Stream_32 Sca;      // count stream to accum (# dot8s per row)
891     Stream_32 Ssa;      // sums (dotNs) from accum
892
893     // output stream processing
894     Stream_32 Sx32; // the output stream for x
895     Stream_64 Sx;   // packed result stream
896
897     // column numbers allow match of A[i][j] with p[j]
898     int j0,j1,j2,j3,j4,j5,j6,j7;
899
900     // A and p inputs to the binary tree dot product unit
901     float a0,a1,a2,a3,a4,a5,a6,a7;
902     float p0,p1,p2,p3,p4,p5,p6,p7;
903
904     // dot product tree multiplier node and adder node outputs
905     float mn0,mn1,mn2,mn3,mn4,mn5,mn6,mn7;
906     float an0,an1,an2,an3,an4,an5,dot8;
907
908     // # bytes: kval, kcol, kptr, b, and x
909     int kvalsz,kcolsz,kptrsz,bsz,xsz;
910
911     float r2nb2nL;  // local copy of residual norm
912     int err;        // needed by the MAC IP cores
```

```
913     int itersL;      // number of iterations
914     int i,j;         // loops
915
916     // number of bytes to be transferred
917     // use SALIGN macro (cg.h) to avoid cache boundary problems
918     kvalsz = SALIGN(knz*sizeof(float),CACHEALIGN);
919     kcolsz = SALIGN(knz*sizeof(int),CACHEALIGN);
920     kptrsz = SALIGN((n+1)*sizeof(int),CACHEALIGN);
921     xsz = SALIGN(n*sizeof(float),CACHEALIGN);
922     bsz = SALIGN(n*sizeof(float),CACHEALIGN);
923
924     // grab kval and kcol (in different GCM banks, per main.c)
925     #pragma src parallel sections
926     {
927     #pragma src section
928     {   // stripe packed kval vector from GCM across four OBMs
929         buffered_dma_gcm(GCM2OBM,PATH_0,kvalL01,
930           MAP_OBM_stripe(1,"A,B,C,D"),kvalA,1,kvalsz);
931     }
932     #pragma src section
933     {   // stripe packed kcol vector from GCM across four OBMs
934         buffered_dma_gcm(GCM2OBM,PATH_1,kcolL01,
935           MAP_OBM_stripe(1,"E,F,G,H"),kcolA,1,kcolsz);
936     }   // section
937     }   // sections
938
939     // grab kptr and the b vector using streams
940     // located in different GCM banks, per main.c
941     #pragma src parallel sections
942     {
943     #pragma src section
944     {   // stream DMA in packed kptr vector from OBCM
```

```
945         streamed_dma_gcm_256(&Skptr256,PORT_TO_STREAM,
946           PATH_0,kptrA,1,(int64_t)kptrsz);
947         stream_term(&Skptr256);  // term required by width converter
948     }
949     #pragma src section
950     {   // convert to 32-bit stream
951         stream_width_256to32_term(&Skptr256,&Skptr32);
952     }
953     #pragma src section
954     {   // put kptr 32-bit stream into BRAM
955         int curr; // current kptr value
956         int i,j;
957         for (i=0; i<kptrsz/sizeof(int); i++) { // pipeline depth: 5
958             get_stream(&Skptr32,&curr);
959             // ordering is 1,0,3,2,5,4,...
960             j=(i%2)?i-1:i+1;  // fix the ordering problem
961             kptrL[j] = curr;
962         }
963     }
964     #pragma src section
965     {   // stream DMA in packed b vector from OBCM
966         streamed_dma_gcm_256(&Sb256,PORT_TO_STREAM,
967           PATH_1,bA,1,(int64_t)bsz);
968         stream_term(&Sb256);  // term required by width converter
969     }
970     #pragma src section
971     {   // convert to 32-bit stream
972         stream_width_256to32_term(&Sb256,&Sb32);
973     }
974     #pragma src section
975     {   // put b vector 32-bit stream into BRAM
976         float curr;
```

```
977         int i,j;
978         for (i=0; i<bsz/sizeof(float); i++) { // pipeline depth: 5
979             get_stream_flt(&Sb32,&curr);
980             // ordering is 1,0,3,2,5,4,...
981             j=(i%2)?i-1:i+1;
982             bL[j] = curr;
983         }
984     }   // section
985     }   // sections
986
987     //-----------------------------------------------------------------
988     // now we have all the data, let's do the cg iteration
989     // use streaming accumulator to sum partial dot products
990     // we iterate till we converge or exceed maximum iterations
991     //-----------------------------------------------------------------
992
993     // do the pre-loop stuff
994     // x0=0, r0=b, p1=b
995     // and precalculate dot(r0,r0)
996     for (i=0; i<n; i++) {
997         xL[i]=0;
998         ri=bL[i];
999         pnxt[i]=ri;
1000        r[i] = ri;
1001        // use to calculate dot(r,r)
1002        // needed for convergence test
1003        fp_mac_32(ri,ri,(i==n-1),1,(i==0),&rTrold,&err);
1004        //          ^  ^      ^   ^    ^    |       |
1005        // r[i] ----+  |      |   |    |    |       |
1006        // r[i] -------+      |   |    |    |       |
1007        // terminate? --------+   |    |    |       |
1008        // accumulate? ------------+    |    |       |
```

```
1009            // reset? ------------------------+    |       |
1010            // output dot(r,r) <-----------------+       |
1011            // output errors <---------------------------+
1012        }
1013
1014        r2nb2nL = EPSILON+1;  // force loop entry
1015        for(itersL=0; ((itersL < MAXITERS)&&(r2nb2nL > EPSILON)); itersL++) {
1016
1017            // copy pnxt to pC* for the current iteration
1018            for (i=0; i<n; i++) {  // pipeline depth: 5
1019                pC0[i] = pnxt[i];  // each leaf node multiplier of the
1020                pC1[i] = pnxt[i];  // binary tree needs a different
1021                pC2[i] = pnxt[i];  // value from the p vector.
1022                pC3[i] = pnxt[i];  // since the BRAMs are single-ported,
1023                pC4[i] = pnxt[i];  // we need 8 independent copies of p
1024                pC5[i] = pnxt[i];  // to avoid an 8-cycle loop-carried
1025                pC6[i] = pnxt[i];  // dependence.
1026                pC7[i] = pnxt[i];
1027
1028                pC8[i] = pnxt[i];  // to calculate dot(p,v)
1029                pC9[i] = pnxt[i];  // to calculate next x
1030                pCa[i] = pnxt[i];  // to calculate next search direction
1031            }
1032
1033            // compute v = Ap
1034            #pragma src parallel sections
1035            {
1036            #pragma src section
1037            {   // send the dot8 vals to accumulator
1038                int i;
1039
1040                for (i=0; i<knz/K; i++) {   // pipeline depth: 63
```

```
1041                                    // all k-groups in each row
1042                    // grab next k matrix values (a8 values)
1043                    split_64to32_flt_flt(kvalL01[i],&a1,&a0); // 2 each
1044                    split_64to32_flt_flt(kvalL23[i],&a3,&a2); // unpack
1045                    split_64to32_flt_flt(kvalL45[i],&a5,&a4); // from
1046                    split_64to32_flt_flt(kvalL67[i],&a7,&a6); // OBM
1047
1048                    // grab next k column pointers (j8 columns)
1049                    split_64to32(kcolL01[i],&j1,&j0);         // 2 each
1050                    split_64to32(kcolL23[i],&j3,&j2);         // unpack
1051                    split_64to32(kcolL45[i],&j5,&j4);         // from
1052                    split_64to32(kcolL67[i],&j7,&j6);         // OBM
1053
1054                    p0 = pC0[j0];          // use the j8 column pointers
1055                    p1 = pC1[j1];          // to grab matching p's (p8)
1056                    p2 = pC2[j2];          // remember, we have k
1057                    p3 = pC3[j3];          // copies of p
1058                    p4 = pC4[j4];          // to avoid memory-based
1059                    p5 = pC5[j5];          // loop-carried dependence
1060                    p6 = pC6[j6];
1061                    p7 = pC7[j7];
1062
1063                    // binary tree pipeline for dot8
1064                    // recall, this delivers a result
1065                    // every clock cycle (after the latency)
1066                    //
1067                    // a0,p0--->*mn0
1068                    //              >+an0
1069                    // a1,p1--->*mn1      \
1070                    //                     >+an4
1071                    // a2,p2--->*mn2      /     \
1072                    //              >+an1        \
```

```
1073                    //  a3 , p3 —>*mn3                  \
1074                    //                                   >+dot8 —>
1075                    //  a4 , p4 —>*mn4                  /
1076                    //                  >+an2          /
1077                    //  a5 , p5 —>*mn5       \        /
1078                    //                        >+an5
1079                    //  a6 , p6 —>*mn6       /
1080                    //                  >+an3
1081                    //  a7 , p7 —>*mn7
1082
1083                    mn0  = a0 * p0;       // leaf nodes (multiplier stage)
1084                    mn1  = a1 * p1;
1085                    mn2  = a2 * p2;
1086                    mn3  = a3 * p3;
1087                    mn4  = a4 * p4;
1088                    mn5  = a5 * p5;
1089                    mn6  = a6 * p6;
1090                    mn7  = a7 * p7;
1091
1092                    an0  = mn0 + mn1;     // first adder stage
1093                    an1  = mn2 + mn3;
1094                    an2  = mn4 + mn5;
1095                    an3  = mn6 + mn7;
1096
1097                    an4  = an0 + an1;     // second adder stage
1098                    an5  = an2 + an3;
1099
1100                    dot8 = an4 + an5;     // root node output adder
1101
1102                    // stream next dot8 to streaming accumulator
1103                    put_stream_flt(&Sva,dot8,1);
1104
```

```
1105            } // for i (next k-group)
1106            stream_term(&Sva);  // termination required by accum
1107        }
1108        #pragma src section
1109        {   // send # of dot8's per row to the accumulator
1110            int i;
1111            int curr,prev; // current & previous to calculate counts
1112            int dot8cnt;   // number of dot8s per row
1113
1114            // note: kptr includes n+1'th item, also
1115            // transfer curr to prev is part of loop control
1116            prev = kptrL[0]; // preload initial prev value
1117            for (i=1; i<=n; prev=curr, i++) {  // pipeline depth: 5
1118                                               // calculate # of
1119                curr = kptrL[i];               // partial dot products
1120                dot8cnt = curr - prev;         // per matrix row, send
1121                put_stream(&Sca,dot8cnt,1);    // to streaming accum
1122            }
1123            stream_term(&Sca);  // termination required by accum
1124        }
1125        #pragma src section
1126        {   // streaming accumulator
1127            stream_fp_accum_strm_counts_32_rr_term(&Sva,&Sca,&Ssa);
1128            //                                      ^     ^    |
1129            // input dot product vals (dot8s) ------+     |    |
1130            // input # of dot8s for each row --------------+    |
1131            // output full dot product stream (dotN) <----------+
1132        }   // section
1133        #pragma src section
1134        {   // finish v = Ap calculation, store in BRAM array
1135            float dotNi;       // the i'th dotN = Ai * xp
1136            int i;
```

```
1137
1138            for (i=0; i<n; i++) {    // pipeline depth: 100
1139                get_stream_flt(&Ssa,&dotNi);      // from accumulator
1140                v[i]= dotNi;
1141
1142                // use a multiply-accumulate to calculate
1143                // dot(p,v)
1144                fp_mac_32(dotNi,pC8[i],(i==n-1),1,(i==0),&pTv,&err);
1145                //          ^     ^      ^      ^   ^    ^     ^
1146                // v[i] ----+     |      |      |   |    |     |
1147                // p[i] ----------+      |      |   |    |     |
1148                // terminate? -----------+      |   |    |     |
1149                // accumulate? ------------------+   |    |     |
1150                // reset? ---------------------------+    |     |
1151                // output dot(p,v) <----------------------+     |
1152                // output errors <-----------------------------+
1153            }
1154        }   // section
1155        }   // sections
1156
1157        alpha = rTrold/pTv;          // step size
1158
1159        for(i=0;i<n;i++) {
1160            xL[i] += alpha * pC9[i];     // next point
1161
1162            // we use the scalar ri to avoid a
1163            // mixed read/write 2-cycle loop, i.e.,
1164            // cannot update an array and then
1165            // read back the updated value in the
1166            // same clock cycle
1167            ri = r[i] - alpha * v[i];    // residual
1168            r[i] = ri;
```

```
1169            // use a multiply-accumulate to calculate new dot(r,r)
1170            fp_mac_32(ri,ri,(i==n-1),1,(i==0),&rTrnew,&err);
1171            //          ^  ^      ^   ^    ^    |       |
1172            // r[i] ---+  |      |   |    |    |       |
1173            // r[i] ------+      |   |    |    |       |
1174            // terminate? -------+   |    |    |       |
1175            // accumulate? ----------+    |    |       |
1176            // reset? --------------------+    |       |
1177            // output dot(r,r) <---------------+       |
1178            // output errors <-------------------------+
1179        }
1180
1181        beta = rTrnew/rTrold;   // projection operator
1182        rTrold = rTrnew;        // for next iteration
1183
1184        for(i=0;i<n;i++) {      // next search direction
1185            pnxt[i] = r[i] + beta * pCa[i];
1186        }
1187        // ||r||/||b|| convergence test
1188        r2nb2nL = sqrt(rTrold) * b2n_1;
1189
1190    } // for itersL... (while not converged)
1191
1192    // converged: send the result back
1193    #pragma src parallel sections
1194    {
1195    #pragma src section
1196    {   // grab x's, pack, and stream them out
1197        float prev,curr;
1198        int64_t packed;
1199        int i;
1200
```

```
1201            prev = xL[0];
1202            // note: transfer to prev is part of loop control
1203            for (i=1; i<xsz/sizeof(float); prev=curr,i++) {  // pipeline depth: 4
1204                curr = xL[i];
1205                if (i%2) { // odd, so combine
1206                    comb_32to64_flt_flt(curr,prev,&packed);
1207                }
1208                put_stream(&Sx,packed,(i%2));  // when it is odd
1209            }
1210        }   // section
1211        #pragma src section
1212        {   // DMA stream out x values in packed stream
1213            streamed_dma_gcm(&Sx,
1214              STREAM_TO_PORT,PATH_0,xA,1,(int64_t)xsz);
1215        }   // section
1216        }   // sections
1217
1218        *iters = itersL;   // iteration count
1219        *r2nb2n = r2nb2nL; // residual norm
1220        return;
1221    }
1222
1223    // Compilation statistics from nohup.out
1224    //
1225    // ##################################################################
1226    // ###############          PLACE AND ROUTE SUMMARY      ############
1227    // ALUTS:                 38,732 / 143,520 ( 27 % )
1228    // Registers:             71,301 / 143,520 ( 50 % )
1229    // Logic utilization:     84,634 / 143,520 ( 59 % )
1230    // M512 rams:             81 / 930 ( 9 % )
1231    // M4K rams:              662 / 768 ( 86 % )
1232    // M-RAMs:                9 / 9 ( 100 % )
```

```
1233 // DSP blocks:           210 / 768 ( 27 % )
1234 // Fast model timing requirements succeeded.
1235 // Slow model timing requirements succeeded.
1236 // ################################################################
1237 //
1238 // Parallel region starting at line 203 has 2 parallel sections.
1239 // Parallel region starting at line 224 has 6 parallel sections.
1240 // Parallel region starting at line 315 has 4 parallel sections.
1241 // Parallel region starting at line 475 has 2 parallel sections.
1242 //
1243 // ################################################################
1244 //
1245 // ##################         INNER LOOP SUMMARY       ############
1246 // loop on line 240:
1247 //     clocks per iteration:    1
1248 //     pipeline depth:          5
1249 //
1250 // loop on line 261:
1251 //     clocks per iteration:    1
1252 //     pipeline depth:          5
1253 //
1254 // loop on line 279:
1255 //     clocks per iteration:    1
1256 //     pipeline depth:         89
1257 //
1258 // loop on line 301:
1259 //     clocks per iteration:    1
1260 //     pipeline depth:          5
1261 //
1262 // loop on line 321:
1263 //     clocks per iteration:    1
1264 //     pipeline depth:         65
```

```
1265 //
1266 // loop on line 398:
1267 //      clocks per iteration:    1
1268 //      pipeline depth:          5
1269 //
1270 // loop on line 419:
1271 //      clocks per iteration:    1
1272 //      pipeline depth:         89
1273 //
1274 // loop on line 439:
1275 //      clocks per iteration:    1
1276 //      pipeline depth:        110
1277 //
1278 // loop on line 466:
1279 //      clocks per iteration:    1
1280 //      pipeline depth:         25
1281 //
1282 // loop on line 485:
1283 //      clocks per iteration:    1
1284 //      pipeline depth:          4
1285 // ##############################################################
```

A.2 ”Tuned” Conjugate Gradient Development

A.2.1 main2.c

```
1286 /* main.c
1287   sparse matrix jacobi iterative solver for SRC-7 hardware
1288   Gerald R. Morris, gerald.r.morris@us.army.mil, 1 Mar 10
1289   refactored by Gerald R. Morris, 7 Apr 2011 to use a
1290   residual norm-based convergence test
1291   Refactored by Jamory D. Hawkins and Anas Alfarra,Dec 2013
1292   to use conjugate gradient as the iteration method
```

```
1293
1294    There are actually two options when building this code
1295    The options are selected via defines in the mvm.h header file
1296    #define SWVER    or   #define HWVER
1297    The first is the software version which employs the standard
1298    nested loop approach to do the calculation serially.
1299    The second is the k-aligned hardware version, which employ loops
1300    wherein the matrix is dealt with in chunks of k elements at a time.
1301    k-alignment requires an exact multiple of k elements in a row
1302    The k-alignment process simply pads the matrix with 0's
1303  */
1304  #include <stdio.h>
1305  #include <stdlib.h>
1306  #include <sys/time.h>
1307  #include <string.h>
1308  #include <stdint.h>
1309  #include <math.h>
1310  #include "mmio.h"
1311  #ifndef CG_H
1312      #include "cg.h"
1313  #endif
1314  #include "sparsekit.h"
1315  #include "usec.h"
1316  #ifndef PARMS_H
1317      #include "parms.h"
1318  #endif
1319
1320  // define this when we want a verbose mode (mostly debug stuff)
1321  //#define VURBOZE
1322
1323  int main (int argc, char *argv[]) {
1324
```

```
1325      // matrices are stored in MatrixMarket coordinate format
1326      // we use the MatrixMarket mmio (I/O) routines
1327      char fname[255];          // MatrixMarket file
1328      FILE *fmm;                // MatrixMarket file pointer
1329      MM_typecode matcode;      // MatrixMarket meta-data
1330      int n;                    // # matrix rows
1331      int nc;                   // # matrix columns
1332      int nnz;                  // # non-zero elements
1333      float *acoo;              // matrix values
1334      int *icoo;                // matrix row indices
1335      int *jcoo;                // matrix column indices
1336
1337      // cg uses compressed sparse row format
1338      float *val;               // CSR sparse matrix (input matrix)
1339      int *col;                 // CSR column index
1340      int *ptr;                 // CSR row pointer
1341
1342      float *x;                 // x vector
1343      float *b;                 // constant vector
1344      float b2n_1;              // (2-norm of b vector)^(-1)
1345      float r2nb2n;             // residual norm(convergence test)
1346
1347      char result[1024];        // capture results
1348      int iters;                // number of iterations needed
1349      FILE *fres;               // results file pointer
1350
1351      double t0,t1,t2,t3;       // to capture wall clock run time
1352      int i,j,k;                // loop indices
1353
1354  #ifdef VURBOZE  // format strings, etc.. when we run in verbose mode
1355      char *f1="\n%5d:    %9.2e     %9.2e     %9.2e     %9.2e\n";
1356      char *f2="          (%4d,%4d)  (%4d,%4d)  (%4d,%4d)  (%4d,%4d)";
```

```
1357      char *f3="\n%5d:%9.2e%9.2e%9.2e%9.2e%6d%5d%5d%5d";
1358      char *r1="                val                              col";
1359      char *r2=" 0         1         2         3         0    1    2    3";
1360      char *r3="-------------------------------------------------------------";
1361      char *indent = "           ";
1362      char *spc="...";
1363      int head = 16;           // 1st few matrix values
1364  #endif // #ifdef VURBOZE
1365
1366      t0 = usec();             // first start time
1367
1368      // matrix file name is specified on the command line
1369      if (argc == 2) {
1370          strcpy(fname,argv[1]);
1371      } else {
1372          fprintf(stderr,"*** Usage: ./cg MM_file\n");
1373          exit (1);
1374      }
1375
1376      // open the input and result files
1377      if ((fmm = fopen(fname, "r")) == NULL) {
1378          fprintf(stderr, "Cannot open file %s\n", fname);
1379          exit(1);
1380      }
1381      fprintf(stderr,"Reading %s\n",fname);
1382      if ((fres = fopen ("result", "w")) == NULL) {
1383          fprintf (stderr, "failed to open file 'result'\n");
1384          exit (1);
1385      }
1386
1387      // get matrix information from MatrixMarket file
1388      mm_read_banner(fmm, &matcode);
```

```
1389        mm_read_mtx_crd_size(fmm, &n, &nc, &nnz);
1390        fprintf(stderr,"   %dx%d %s with %d nonzero values\n",
1391          n,nc,mm_typecode_to_str(matcode),nnz);
1392
1393        // allocate coordinate format arrays now that sizes are known
1394        acoo = (float *) malloc(nnz * sizeof(float));
1395        icoo = (int *) malloc(nnz * sizeof(int));
1396        jcoo = (int *) malloc(nnz * sizeof(int));
1397
1398        // read MatrixMarket coordinate-format matrix data
1399        for (i=0; i<nnz; i++) {
1400            fscanf(fmm, "%d %d %g\n", &icoo[i], &jcoo[i], &acoo[i]);
1401            icoo[i]--;  // adjust from 1-based to 0-based for C arrays
1402            jcoo[i]--;
1403        }
1404        fclose(fmm);
1405
1406    #ifdef VURBOZE // dump up to head elements of input matrix
1407        printf("%s%s","COO:       ",r3);
1408        for (i=0; i<MIN(nnz,3*head); i+=4) {
1409            printf(f1,
1410              i,acoo[i],acoo[i+1],acoo[i+2],acoo[i+3]);
1411            printf(f2,icoo[i],jcoo[i],icoo[i+1],jcoo[i+1],
1412              icoo[i+2],jcoo[i+2],icoo[i+3],jcoo[i+3]);
1413        }
1414        printf("    %s\n",spc);
1415        printf("%s%s\n",indent,r3);
1416    #endif // #ifdef VURBOZE
1417
1418        // allocate CSR array and convert to CSR format
1419        val = (float *) malloc(nnz * sizeof(float));
1420        col = (int *) malloc(nnz * sizeof(int));
```

```
1421        ptr = (int *) malloc((n+1) * sizeof(int));
1422        coocsr(n,nnz,acoo,icoo,jcoo,val,col,ptr);   // SPARSKIT routine
1423
1424        // have CSR data so deallocate MatrixMarket arrays
1425        free(acoo);
1426        free(icoo);
1427        free(jcoo);
1428
1429    #ifdef VURBOZE  // show the CSR format matrix data
1430        // val, and col vectors
1431        printf("%s%s\n",indent,r1);
1432        printf("%s%s\n",indent,r2);
1433        printf("%s%s","CSR:       ",r3);
1434        for (i=0; i<MIN(nnz,3*head); i+=4) {
1435            printf(f3,
1436              i,val[i],val[i+1],val[i+2],val[i+3],
1437                 col[i],col[i+1],col[i+2],col[i+3]);
1438        }
1439        printf("     %s\n",spc);
1440
1441        // and ptr array  (sideways)
1442        printf("%s%s\n",indent,r3);
1443        printf("    i:      ");
1444        for (i=0; i<=MIN(n,head); i++) {
1445            printf("%5d",i);
1446        }
1447        printf("  %s\n",spc);
1448        printf("    ptr:    ");
1449        for (i=0; i<=MIN(n,head); i++) {
1450            printf("%5d",ptr[i]);
1451        }
1452        printf("  %s\n",spc);
```

```
1453        printf("%s%s\n",indent,r3);
1454    #endif // #ifdef VURBOZE
1455
1456        // allocate solution and constant vector
1457        x = (float *) malloc (n * sizeof(float));
1458        b = (float *) malloc (n * sizeof(float));
1459
1460        // calc constant vector b assuming x[i] = 1000.0
1461        // we'll dump the b vector to our result file
1462        // also calculate 2-norm of b vector
1463        b2n_1 = 0.0;
1464        for (i=0; i<n; i++) {
1465            b[i] = 0.0;
1466            for (j=ptr[i]; j<ptr[i+1]; j++) {    // sum(aij*xj)
1467                b[i]+=val[j]*1000.0;
1468            }
1469            b2n_1 += b[i]*b[i];                    // sum(b[i]^2)
1470            fprintf(fres,"b[%d] = %12.10e\n",i,b[i]);
1471        }
1472        b2n_1 = 1/sqrt(b2n_1);          // (2-norm of b vector)^(-1)
1473
1474        // all of that just to get our matrix and constant vectors!
1475        // now we do the software or hardware version of cg
1476
1477        // do the cg 5 times to get good average
1478        t1 = usec();              // first stop time, second start time
1479        for(k=0;k<5;k++) { // run it 5 times to get a good average
1480            // initialize the x vector to all zeros
1481            // basically we start "guessing" at x[i] = 0
1482            for (i=0; i<n; i++) {
1483                x[i] = 0.0;
1484            }
```

```
1485            cg(val,col,ptr,b,n,nnz,x,b2n_1,&r2nb2n,&iters); // solve for x
1486        }
1487        t2 = usec();                                        // second stop time, third start time
1488
1489        // save results
1490        for (i=0; i<n; i++) {
1491            fprintf(fres, "x[%d] = %12.10e\n",i,x[i]);
1492        }
1493
1494        // close output file and release memory
1495        fclose(fres);
1496        free(val);
1497        free(col);
1498        free(ptr);
1499
1500        // report to stdout and quit
1501        t3 = usec();              // third stop time
1502        sprintf(result,"%s \t %10d \t %6d \t %12.10f \t %12.10f",
1503          fname,nnz,iters,r2nb2n,
1504          (t1 - t0) + (t2 - t1)/5 + (t3 - t2));
1505        puts(result);
1506        if(iters<MAXITERS) {
1507            exit(0);
1508        } else {
1509            sprintf(result,"*** Maximum iterations (%d) exceeded...",MAXITERS);
1510            puts(result);
1511            exit(1);
1512        }
1513    }
```

A.2.2 cg2.h

```
1514    /* cg.h
```

```
1515     J. Hawkins, A. Alfarra, and G. Morris, Dec 2013
1516     prototype for sparse matrix conjugate gradient
1517     J. Hawkins and G. Morris, Jan 2014
1518     now only matrix vector multiply is executed in hardware
1519     so there are two versions: software and hardware
1520  */
1521  #ifndef CG_H
1522      #define CG_H
1523  #endif
1524
1525  // cg iteration function
1526  void cg(
1527    float val[],    // CSR sparse matrix values
1528    int col[],      // CSR column indices
1529    int ptr[],      // CSR row pointers
1530    float b[],      // known constant vector
1531    int n,          // matrix order
1532    int nnz,        // number of non-zeros
1533    float x[],      // solve for x
1534    float b2n_1,    // (2-norm of b vector)^(-1)
1535    float *r2nb2n, // ratio of 2-norms (convergence test)
1536    int *iters      // iterations to solve or give up
1537  );
```

A.2.3 cg2.c

```
1538  /* cg.c
1539     J. Hawkins, A. Alfarra, and G. Morris, Dec 2013
1540     software version of sparse matrix conjugate gradient
1541
1542     refactored Jan 2014 to accomodate matrix vector multiply
1543  */
1544  #include <stdlib.h>
```

```
1545 // #include <math.h>
1546 #ifndef CG_H
1547     #include "cg.h"
1548 #endif
1549 #ifndef MVM_H
1550     #include "mvm.h"
1551 #endif
1552 #ifndef PARMS_H
1553     #include "parms.h"
1554 #endif
1555
1556 void cg(
1557   float val[],    // CSR sparse matrix values
1558   int col[],      // CSR column index
1559   int ptr[],      // CSR row pointer
1560   float b[],      // known constant vector
1561   int n,          // matrix order
1562   int nnz,        // number of non-zeros
1563   float x[],      // solve for x
1564   float b2n_1,    // (2-norm of b vector)^(-1)
1565   float *r2nb2n, // residual norm (convergence test)
1566   int *iters) {  // iterations to solve or give up
1567
1568     /* x_0 = 0 so Ax_0 = 0 , ergo vector Ax not needed
1569     float *Ax;              // for initial search direction */
1570
1571     // p, v vectors allocated for either software or hardware
1572     float *p;               // search direction
1573     float *v;               // avoid multiple Ap mvm
1574
1575     float *r;               // residual vector
1576     float alpha;            // step size
```

```
1577      float rTrold;             // dot(r(k-1),r(k-1))
1578      float rTrnew;             // dot(r(k),r(k))
1579      float beta;               // projection operator
1580      float normr;              // 2-norm of r (||r||)
1581      float delta;              // residual norm = ||r||/||b||
1582      int iterslcl = 1;         // local iteration index
1583      float sum;                // multi-use accumulator
1584      int i,j;                  // index variables
1585
1586  #ifdef HWVER  // if it's the hardware version, we need these
1587                // for k-aligned CSR data
1588      float *kval;              // CSR matrix k-aligned values
1589      int *kcol;                // CSR k-aligned column index
1590      int *kptr;                // CSR k-aligned row pointer
1591      int knz;                  // k-aligned # of non-zeros
1592      int kvalsz;               // allocation size of kval (bytes)
1593      int kcolsz;               // allocation size of kcol
1594      int kptrsz;               // allocation size of kptr
1595      int psz;                  // allocation size of vectors
1596      int vsz;
1597      int gotA = 0;             // true when A is already on FPGA
1598      // for MAP allocation and usage
1599      int nmaps = 1;            // the number of MAPs needed
1600      int mapnum = 0;            // first MAP number is 0
1601
1602      // we'll allocate segments using both OBCM banks
1603      // and marshall the MAP data in those banks (speed)
1604      gcm_seg_desc_t *sd1;      // OBCM segment descriptor pointers
1605      gcm_seg_desc_t *sd2;      // for bank 1 and bank 2
1606      uint64_t sd1sz;           // segment sizes
1607      uint64_t sd2sz;
1608      // address of the MAP data allocated from the 2 segments above
```

```
1609        gcm_addr_t kvalA;           // OBCM address of CSR matrix values
1610        gcm_addr_t kcolA;           // OBCM address of CSR column indices
1611        gcm_addr_t kptrA;           // OBCM address of CSR row pointers
1612        gcm_addr_t pA;              // OBCM address of current search direction
1613        gcm_addr_t vA;              // OBCM address of v=ap
1614
1615        // HWVER: allocate and load the matrix into cache-aligned buffers.
1616        // as part of software/hardware codesign tradeoffs the
1617        // k-alignment of A will be done here in software
1618        // the MAP wants the CSR vectors to be aligned on cache
1619        // boundaries.  We can't place this restriction on upper-level
1620        // callers, so we will marshall the data here into
1621        // properly aligned arrays.  Since we have to do the memcopy
1622        // anyway, we'll just go ahead and do the k-alignment needed
1623        // by our MAP routine here in software.
1624
1625        // figure out the growth in kval and kcol due to the
1626        // k-alignment processing, i.e., calculate knz
1627        knz = 0;
1628        for (i=0; i<n; i++) {           // loop across all matrix rows
1629            for (j=ptr[i]; j<ptr[i+1]; j++) {
1630                knz++;                  // count # non-zeros
1631            }
1632            while ((knz%K) != 0) {      // pad with 0's if necessary
1633                knz++;                  // to make knz a multiple of K
1634            }
1635        }
1636
1637        // now we have the knz value, so we can calculate array sizes
1638        // use SALIGN macro (mvm.h) to avoid cache boundary problems
1639        kvalsz = SALIGN(knz*sizeof(float),CACHEALIGN);
1640        kcolsz = SALIGN(knz*sizeof(int),CACHEALIGN);
```

```
1641    kptrsz = SALIGN((n+1)*sizeof(int),CACHEALIGN);
1642    psz = SALIGN(n*sizeof(float),CACHEALIGN);
1643    vsz = SALIGN(n*sizeof(float),CACHEALIGN);
1644
1645    // allocate arrays for the k-aligned CSR matrix
1646    kval = (float*)Cache_Aligned_Allocate (kvalsz);
1647    kcol = (int*)  Cache_Aligned_Allocate (kcolsz);
1648    kptr = (int*)  Cache_Aligned_Allocate (kptrsz);
1649
1650    // allocate vectors for search direction and v=ap result
1651    // these are allocated differently for software
1652    p = (float*)Cache_Aligned_Allocate (psz);
1653    v = (float*)Cache_Aligned_Allocate (vsz);
1654
1655    // also need OBCM bank segment descriptors
1656    // to speed up MAP processing, place data in separate banks
1657    // kval and kptr go into OBCM bank 1
1658    // kcol, p, and v go into OBCM bank 2
1659    // per SRC7CPEGuide, must start on 8-byte
1660    // boundaries and be allocated in length that is a multiple of 8
1661    // this is already guaranteed because of our cache aligned sizes
1662    sd1sz = (uint64_t)kvalsz + (uint64_t)kptrsz;
1663    sd2sz = (uint64_t)kcolsz + (uint64_t)psz + (uint64_t)vsz;
1664    if (gcm_allocate_seg_by_bank(sd1sz, 1, &sd1)) {
1665        fprintf(stderr,"Bank 1 OBCM segment allocation failed\n");
1666        exit(1);
1667    }
1668    if (gcm_allocate_seg_by_bank(sd2sz, 2, &sd2)) {
1669        fprintf(stderr,"Bank 2 OBCM segment allocation failed\n");
1670        exit(1);
1671    }
1672    // now allocate the buffer addresses in OBCM segments
```

```
1673      if (!(kvalA = gcm_allocate_from_seg(kvalsz,sd1))) {
1674          fprintf(stderr,"OBCM allocation for kval failed\n");
1675          exit(1);
1676      }
1677      if (!(kcolA = gcm_allocate_from_seg(kcolsz,sd2))) {
1678          fprintf(stderr,"OBCM allocation for kcol failed\n");
1679          exit(1);
1680      }
1681      if (!(kptrA = gcm_allocate_from_seg(kptrsz,sd1))) {
1682          fprintf(stderr,"OBCM allocation for kptr failed\n");
1683          exit(1);
1684      }
1685      if (!(pA = gcm_allocate_from_seg(psz,sd2))) {
1686          fprintf(stderr,"OBCM allocation for p failed\n");
1687          exit(1);
1688      }
1689      if (!(vA = gcm_allocate_from_seg(vsz,sd2))) {
1690          fprintf(stderr,"OBCM allocation for v failed\n");
1691          exit(1);
1692      }
1693
1694      // k-align the kval, kcol, and kptr arrays
1695      // the meaning of kptr is slightly different than ptr
1696      // recall, MAP will be grabbing kcol and kval elements
1697      // K per clock cycle, so kptr expresses things in groups of K
1698      knz = 0;
1699      for (i=0; i<n; i++) {                    // all matrix rows
1700          kptr[i] = knz/K;                     // k-aligned ptr value
1701          for (j=ptr[i]; j<ptr[i+1]; j++) {
1702              kval[knz] = val[j];
1703              kcol[knz++] = col[j];
1704          }
```

```
1705            while ((knz%K) != 0) {              // pad with 0's
1706                kval[knz] = 0.0;                // to fill a group of K
1707                kcol[knz++] = 0;                // since a[i][j]=0,
1708            }                                   // the p used is negligent
1709        }
1710        kptr[i] = knz/K;                        // similar to ptr[n+1]
1711
1712    #ifdef VURBOZE  // show the k-aligned sizes and data
1713        printf("words:  n,knz = %d,%d\n",n,knz);
1714        printf("\nbytes:  |kval|=%d,|kcol|=%d,|kptr|=%d,|vec|=%d\n",
1715          kvalsz,kcolsz,kptrsz,xsz);
1716
1717        // show the k-aligned kval, and kcol vectors
1718        printf("\n%s%s\n",indent,k1);
1719        printf("%s%s\n",indent,r2);
1720        printf("%s%s","kCSR:      ",r3);
1721        for (i=0; i<MIN(knz,3*head); i+=4) {
1722            printf(f3,
1723              i,kval[i],kval[i+1],kval[i+2],kval[i+3],
1724                kcol[i],kcol[i+1],kcol[i+2],kcol[i+3]);
1725        }
1726        printf("   %s\n",spc);
1727
1728        // show the k-aligned kptr array (sideways)
1729        printf("\n   i:     ");
1730        for (i=0; i<=MIN(n,head); i++) {
1731            printf("%4d",i);
1732        }
1733        printf("  %s\n",spc);
1734        printf("   kptr:  ");
1735        for (i=0; i<=MIN(n,head); i++) {
1736            printf("%4d",kptr[i]);
```

```
1737        }
1738        printf("  %s\n",spc);
1739        printf("  %s\n",spc);
1740        printf("%s%s\n",indent,r3);
1741    #endif // #ifdef VURBOZE
1742
1743        if (map_allocate(nmaps)) {  // allocate the MAP
1744            fprintf(stderr,"Could not allocate MAP.  Bye!\n");
1745            exit(1);
1746        }
1747
1748    #else // SWVER
1749
1750        // p and v vectors are allocated differently for software
1751        p = (float *)malloc(n*sizeof(float));
1752        v = (float *)malloc(n*sizeof(float));
1753    #endif // HWVER
1754
1755
1756        // allocate our residual vector
1757        r = (float *)malloc(n*sizeof(float));
1758        // Ax = (float *)malloc(n*sizeof(float));
1759
1760        // how to calculate Ax_0 if were necessary
1761        /* x_0 = 0 so Ax_0 = 0, ergo no calculation
1762        for(i=0;i<n;i++) {
1763            sum = 0; // recall A is sparse
1764            for(j=ptr[i];j<ptr[i+1];j++) {
1765                sum += val[j]*x[col[j]];
1766            }
1767            Ax[i]=sum;
1768        }
```

```
1769        */
1770
1771        // algorithms lines 2,3,5,and 7
1772        // recall Ax_0=0 so p1=r0=b
1773        // 1st A-orthogonal search direction (p1)
1774        // 0th residual (r0 = p1)
1775        // rTrold = dot(r0,r0)
1776        rTrold = 0;
1777        for(i=0;i<n;i++) {
1778            x[i] = 0;              // want x_0 = 0
1779            p[i] = b[i];           // p1 = b-Ax_0
1780            r[i] = p[i];           // r0 = p1
1781            rTrold += r[i]*r[i]; // dot(r(0),r(0))
1782        }
1783
1784        delta = EPSILON + 1.0;  // force loop entry
1785        // loop until converge or max iters exceeded
1786        while((delta>EPSILON)&&(iterslcl<MAXITERS)) {
1787
1788            // alg line 10: need Ap twice, so calc once
1789    #ifndef HWVER
1790            // SWVER:
1791            mvm(val,col,ptr,p,n,v);
1792
1793    #else
1794            // HWVER:
1795            if(!gotA) { // only copy A one time
1796                // copy data to the OBCM buffers
1797                gcm_cp_to(kvalA, kval, kvalsz);
1798                gcm_cp_to(kcolA, kcol, kcolsz);
1799                gcm_cp_to(kptrA, kptr, kptrsz);
1800            }
```

```
1801            gcm_cp_to(pA, p, psz);    // always need p
1802            mvm(kvalA, kcolA, kptrA, pA, n, knz, vA, gotA, mapnum);
1803            gotA = 1;
1804            gcm_cp_from(vA, v, vsz); // copy result back from OBCM buffer
1805
1806    #endif // HWVER
1807
1808            // alg 11: calc step size (alpha)
1809            sum = 0;
1810            for(i=0;i<n;i++) {
1811                sum += p[i]*v[i];
1812            }
1813            alpha = rTrold/sum;
1814
1815            // alg 12: calc next x and
1816            // alg 16: residual vector and
1817            // alg 18: dot (r,r)
1818            rTrnew = 0;
1819            for(i=0;i<n;i++) {
1820                x[i] += alpha*p[i];
1821                r[i] -= alpha*v[i];
1822                rTrnew += r[i]*r[i];
1823            }
1824            normr = sqrt(rTrnew); // ||r||, part of alg 22
1825
1826            beta = rTrnew/rTrold; // alg 19: projection operator
1827
1828            rTrold = rTrnew;      // alg 20: for next iteration
1829
1830            // alg 21: calc next search direction
1831            for(i=0;i<n;i++) {
1832                p[i] = r[i]+beta*p[i];
```

```
1833            }
1834
1835            delta = normr * b2n_1;      // alg 22: residual norm
1836            iterslcl++;                 // next iteration
1837
1838        } // end while
1839
1840        *iters = iterslcl;              // report iterations
1841        *r2nb2n = delta;                // and residual norm
1842
1843        // release vectors
1844        /* x_0 = 0 so Ax_0 = 0, ergo no Ax vector
1845        free(Ax); */
1846        free(r);
1847
1848             // release all the stuff that was allocated
1849    #ifndef HWVER
1850        free(p);
1851        free(v);
1852    #else
1853        // deallocate the MAP
1854        if (map_free(nmaps)) {
1855            fprintf(stderr,"Could not deallocate MAP.  Bye!\n");
1856            exit(1);
1857        }
1858        // arrays
1859        Cache_Aligned_Free((char*)kval);
1860        Cache_Aligned_Free((char*)kcol);
1861        Cache_Aligned_Free((char*)kptr);
1862        Cache_Aligned_Free((char*)p);
1863        Cache_Aligned_Free((char*)v);
1864        // OBCM buffers
```

```
1865        gcm_free(kvalA);
1866        gcm_free(kcolA);
1867        gcm_free(kptrA);
1868        gcm_free(pA);
1869        gcm_free(vA);
1870        // OBCM segments
1871        gcm_free_seg(sd1);
1872        gcm_free_seg(sd2);
1873    #endif // HWVER
1874
1875    } // end cg
```

A.2.4 parms.h

```
1876     /* parms.h
1877        J. Hawkins and G. Morris, Jan 2014
1878        parameters needed for all dependent files
1879     */
1880    #ifndef PARMS_H
1881        #define PARMS_H
1882    #endif
1883
1884    // sometimes MIN and MAX macros already defined
1885    #ifndef MIN
1886        #define MIN(x,y) (((x)<(y)?(x):(y)))
1887        #define MAX(x,y) (((x)>(y)?(x):(y)))
1888    #endif
1889
1890    // maximum iterations, convergence criterion, max n
1891    #define MAXITERS (100000)
1892    #define EPSILON (1e-6)
1893    #define NMAX (8192)
```

A.2.5 mvm.h

```
1894 /* mvm.h
1895    J. Hawkins and G. Morris, Jan 2014
1896    now only matrix vector multiply is executed in hardware
1897    so there are two versions: software and hardware
1898 */
1899 #ifndef MVM_H
1900     #define MVM_H
1901 #endif
1902
1903 #define HWVER
1904 #ifdef HWVER
1905     #include <libmap.h>
1906 #endif
1907
1908 // width of the binary tree
1909 #define K (8)
1910
1911 #ifndef HWVER
1912
1913     // SWVER: the software mvm function
1914     void mvm(
1915       float val[],   // CSR sparse matrix values
1916       int col[],     // CSR column indices
1917       int ptr[],     // CSR row pointers
1918       float p[],     // current search direction
1919       int n,         // matrix order
1920       float v[]      // v = ap
1921     );
1922
1923 #else
1924
```

```
1925      // HWVER: the hardware mvm function
1926      void mvm(
1927        gcm_addr_t kvalA, // GCM address of CSR matrix values
1928        gcm_addr_t kcolA, // GCM address of CSR column indices
1929        gcm_addr_t kptrA, // GCM address of CSR row pointers
1930        gcm_addr_t pA,    // GCM address of curr search direction
1931        int n,            // number of matrix rows and columns
1932        int knz,          // k-aligned number of non-zeros
1933        gcm_addr_t vA,    // GCM address of v = ap
1934        int gotA,         // true when A is already on FPGA
1935        int mapnum);      // MAP number required by Carte
1936
1937      // cache alignment (% more /proc/cpuinfo)
1938      #define CACHEALIGN (128)
1939
1940      // return next integer multiple of sz >= p
1941      #define SALIGN(p,sz) ((((p)%(sz))==0)?((p)):((p)+(sz)-((p)%(sz))))
1942
1943  #endif // HWVER
```

A.2.6 mvm.c

```
1944  /* mvm.c
1945     J. Hawkins and G. Morris, Jan 2014
1946     now only matrix vector multiply is executed in hardware
1947     software version
1948  */
1949  #ifndef MVM_H
1950      #include "mvm.h"
1951  #endif
1952
1953  void mvm(
1954    float val[],   // CSR sparse matrix values
```

```
1955    int col[],      // CSR column indices
1956    int ptr[],      // CSR row pointers
1957    float p[],      // current search direction
1958    int n,          // matrix order
1959    float v[]) {    // v = ap
1960
1961      float sum;    // accumulator
1962      int i,j;      // index variables
1963
1964      // need Ap twice, so calc once
1965      for(i=0;i<n;i++) {
1966          sum = 0; // recall A is sparse
1967          for(j=ptr[i];j<ptr[i+1];j++) {
1968              sum += val[j]*p[col[j]];
1969          }
1970          v[i] = sum;
1971      }
1972  } // end mvm
```

A.2.7 mvm.mc

```
1973  // mvm.mc
1974  // sparse matrix vector multiplication,
1975  // Jerry Morris & Jamory Hawkins, Jan 2014
1976  // partly based on Morris' April 2011 Jacobi solver
1977  // this is the hardware version that runs on the SRC-7 MAP
1978
1979  #include <libmap.h>
1980  #include "mvm.h"
1981  #ifndef PARMS_H
1982      #include "parms.h"
1983  #endif
1984
```

111

```
1985 void mvm(
1986     gcm_addr_t kvalA, // GCM address of CSR matrix values
1987     gcm_addr_t kcolA, // GCM address of CSR column indices
1988     gcm_addr_t kptrA, // GCM address of CSR row pointers
1989     gcm_addr_t pA,    // GCM address of curr search direction
1990     int n,            // number of matrix rows and columns
1991     int knz,          // k-aligned number of non-zeros
1992     gcm_addr_t vA,    // GCM address of v = ap
1993     int gotA,         // true when A is already on FPGA
1994     int mapnum) {      // MAP number required by Carte
1995
1996     // k-striped local copy of k-aligned matrix values (kval) in OBM
1997     // 2-pack (0,1)
1998     // 2-pack (2,3)
1999     // 2-pack (4,5)
2000     // 2-pack (6,7)
2001     OBM_BANK_A(kvalL01, int64_t, MAX_OBM_SIZE)
2002     OBM_BANK_B(kvalL23, int64_t, MAX_OBM_SIZE)
2003     OBM_BANK_C(kvalL45, int64_t, MAX_OBM_SIZE)
2004     OBM_BANK_D(kvalL67, int64_t, MAX_OBM_SIZE)
2005
2006     // k-striped local copy of k-aligned matrix columns (kcol) in OBM
2007     // 2-pack (0,1)
2008     // 2-pack (2,3)
2009     // 2-pack (4,5)
2010     // 2-pack (6,7)
2011     OBM_BANK_E(kcolL01, int64_t, MAX_OBM_SIZE)
2012     OBM_BANK_F(kcolL23, int64_t, MAX_OBM_SIZE)
2013     OBM_BANK_G(kcolL45, int64_t, MAX_OBM_SIZE)
2014     OBM_BANK_H(kcolL67, int64_t, MAX_OBM_SIZE)
2015
2016     // all other vectors go into BRAM on the Altera FPGA
```

```
2017    // we play some games to balance use of the BRAM memory.
2018    // EP2S180 FPGA has 768 M4K RAM, 930 M512 RAM, and 9 MRAM blocks.
2019    // we consume 662 of 768 M4K BRAMs, 81 of 930 M512 BRAMs,
2020    // and 9 of 9 MRAM blocks.  Basically we declare some arrays
2021    // bigger than they need to be so that Carte puts them in
2022    // MRAM rather than M4K.  see SRC-7 Carte C Programming
2023    // Environment Guide for additional details.
2024
2025    int kptrL[2*NMAX];   // local copies of kptr
2026
2027    // the dot8 tree needs independent copies of p
2028    // to avoid multiple-cycle memory accesses
2029    float pC0[NMAX];        // the p copy (pC*) are the
2030    float pC1[NMAX];        // p inputs for each leaf node
2031    float pC2[NMAX];        // of a size k=8 dot product unit
2032    float pC3[2*NMAX];      // the dot8 is fully pipelined
2033    float pC4[2*NMAX];      // the array A inputs to the dot8
2034    float pC5[2*NMAX];      // come from the striped OBM banks
2035    float pC6[2*NMAX];
2036    float pC7[2*NMAX];
2037
2038    // input vector stream processing is as follows:
2039    // 1.  DMAstream in 264-bit packed values from GCM
2040    // 2.  stream width convert into 32-bit stream
2041    // 4.  unload single 32-bit stream into local BRAM arrays
2042    Stream_256 Skptr256;              // kptr streams
2043    Stream_32 Skptr32;
2044    Stream_256 Sp256_1;                 // p streams
2045    Stream_256 Sp256_2;                 // p streams
2046    Stream_32 Sp32_1;                   // the streaming accumulator bears explanation
2047    Stream_32 Sp32_2;                   // the streaming accumulator bears explanation
2048
```

```
2049    //
2050    // suppose I have 3 vectors: a=(1,2,3), b=(4,5), c=(6,7,8,9)
2051    // and I want the three vector sums: sa=1+2+3=6, sb=9, sc=30
2052    // further suppose the vector data is being delivered as
2053    // a single stream, vals = 1,2,3,4,5,6,7,8,9
2054    // and I have another stream, cnts = 3,2,4 (sizes of a, b, c)
2055    // I use the streaming accumulator to produce
2056    // a third stream, sums = 6,9,30.
2057    // Here's an elided pseudo-code of the four parallel sections
2058    // that one would use to accomplish this task
2059    //
2060    // pars {
2061    // par { // stream out values
2062    //     for i=1,9 {
2063    //         put_stream(&Sva, vals[i])
2064    //     }
2065    // }
2066    // par {  // stream out counts
2067    //     for i=1,3 {
2068    //         put_stream(&Sca,cnts[i]))
2069    //     }
2070    // par { // eat vals, cnts and stream out sums
2071    //     stream_fp_accum_strm_counts_32_rr_term(&Sva,&Sca,&Ssa)
2072    // }
2073    // par { // save sums
2074    //     for i=1,3 {
2075    //         get_stream(&Ssa,&sums[i]);
2076    // }
2077    // }
2078    //
2079    // in our case accumulator stream processing is as follows:
2080    // 1.  compute dot8 patial dot products: stream dot8 vals to accum
```

```
2081        // 2.  compute number of dot8s per row: stream cnts to accum
2082        // 3.  accum sums dot8s & outputs n full dot products v[i]
2083        // 4.  pack the v[i] and stream up to OBCM
2084        //
2085        Stream_32 Sva;      // value stream (dot8s) to accum
2086        Stream_32 Sca;      // count stream to accum (# dot8s per row)
2087        Stream_32 Ssa;      // sums (dotNs)= v[i] from accum
2088
2089        // output stream processing
2090        Stream_32 Sv32; // the output stream for v
2091        Stream_64 Sv;   // packed result stream
2092
2093        // column numbers allow match of A[i][j] with p[j]
2094        int j0,j1,j2,j3,j4,j5,j6,j7;
2095
2096        // A and p inputs to the binary tree dot product unit
2097        float a0,a1,a2,a3,a4,a5,a6,a7;
2098        float p0,p1,p2,p3,p4,p5,p6,p7;
2099
2100        // dot product tree multiplier node and adder node outputs
2101        float mn0,mn1,mn2,mn3,mn4,mn5,mn6,mn7;
2102        float an0,an1,an2,an3,an4,an5,dot8;
2103
2104        // # bytes: kval, kcol, kptr, p, and v
2105        int kvalsz,kcolsz,kptrsz,psz,vsz;
2106
2107        int i,j;          // loops
2108
2109        // number of bytes to be transferred
2110        // use SALIGN macro (mvm.h) to avoid cache boundary problems
2111        psz = SALIGN(n*sizeof(float),CACHEALIGN);
2112        vsz = SALIGN(n*sizeof(float),CACHEALIGN);
```

```
2113
2114    if(!gotA) { // only need to bring in A one time
2115        kvalsz = SALIGN(knz*sizeof(float),CACHEALIGN);
2116        kcolsz = SALIGN(knz*sizeof(int),CACHEALIGN);
2117        kptrsz = SALIGN((n+1)*sizeof(int),CACHEALIGN);
2118
2119        // grab kval and kcol (in different GCM banks, per cg.c)
2120        #pragma src parallel sections
2121        {
2122        #pragma src section
2123        {   // stripe packed kval vector from GCM across four OBMs
2124            buffered_dma_gcm(GCM2OBM,PATH_0,kvalL01,
2125              MAP_OBM_stripe(1,"A,B,C,D"),kvalA,1,kvalsz);
2126        }
2127        #pragma src section
2128        {   // stripe packed kcol vector from GCM across four OBMs
2129            buffered_dma_gcm(GCM2OBM,PATH_1,kcolL01,
2130              MAP_OBM_stripe(1,"E,F,G,H"),kcolA,1,kcolsz);
2131        }   // section
2132        }   // sections
2133
2134        // grab kptr and p using streams
2135        // located in the GCM bank, per cg.c
2136        #pragma src parallel sections
2137        {
2138        #pragma src section
2139        {   // stream DMA in packed kptr vector from OBCM
2140            streamed_dma_gcm_256(&Skptr256,PORT_TO_STREAM,
2141              PATH_0,kptrA,1,(int64_t)kptrsz);
2142            stream_term(&Skptr256);  // term required by width converter
2143        }
2144        #pragma src section
```

```
2145        {   // convert to 32-bit stream
2146            stream_width_256to32_term(&Skptr256,&Skptr32);
2147        }
2148        #pragma src section
2149        {   // put kptr 32-bit stream into BRAM
2150            int curr; // current kptr value
2151            int i,j;
2152            for (i=0; i<kptrsz/sizeof(int); i++) { // pipeline depth: 5
2153                get_stream(&Skptr32,&curr);
2154                // ordering is 1,0,3,2,5,4,...
2155                j=(i%2)?i-1:i+1;  // fix the ordering problem
2156                kptrL[j] = curr;
2157            }
2158        } // section
2159        #pragma src section
2160        {   // stream DMA in packed p vector from OBCM
2161            streamed_dma_gcm_256(&Sp256_1,PORT_TO_STREAM,
2162              PATH_1,pA,1,(int64_t)psz);
2163            stream_term(&Sp256_1);  // term required by width converter
2164        }
2165        #pragma src section
2166        {   // convert to 32-bit stream
2167            stream_width_256to32_term(&Sp256_1,&Sp32_1);
2168        }
2169        #pragma src section
2170        {   // put p vector 32-bit stream into BRAM
2171            float curr;
2172            int i,j;
2173            for (i=0; i<psz/sizeof(float); i++) { // pipeline depth: 5
2174                get_stream_flt(&Sp32_1,&curr);
2175                // ordering is 1,0,3,2,5,4,...
2176                j=(i%2)?i-1:i+1;
```

```
2177                // remember we need 8 copies of p for the dot product unit
2178                pC0[j] = curr;
2179                pC1[j] = curr;
2180                pC2[j] = curr;
2181                pC3[j] = curr;
2182                pC4[j] = curr;
2183                pC5[j] = curr;
2184                pC6[j] = curr;
2185                pC7[j] = curr;
2186            }
2187        } // section
2188        } // sections
2189    } else {      // only need p
2190        // grab p using streams
2191        // located in the GCM bank, per cg.c
2192
2193        #pragma src parallel sections
2194        {
2195        #pragma src section
2196        {   // stream DMA in packed p vector from OBCM
2197            streamed_dma_gcm_256(&Sp256_2,PORT_TO_STREAM,
2198              PATH_1,pA,1,(int64_t)psz);
2199            stream_term(&Sp256_2);  // term required by width converter
2200        }
2201        #pragma src section
2202        {   // convert to 32-bit stream
2203            stream_width_256to32_term(&Sp256_2,&Sp32_2);
2204        }
2205        #pragma src section
2206        {   // put p vector 32-bit stream into BRAM
2207            float curr;
2208            int i,j;
```

```
2209            for (i=0; i<psz/sizeof(float); i++) { // pipeline depth: 5
2210                get_stream_flt(&Sp32_2,&curr);
2211                // ordering is 1,0,3,2,5,4,...
2212                j=(i%2)?i-1:i+1;
2213                // remember we need 8 copies of p for the dot product unit
2214                pC0[j] = curr;
2215                pC1[j] = curr;
2216                pC2[j] = curr;
2217                pC3[j] = curr;
2218                pC4[j] = curr;
2219                pC5[j] = curr;
2220                pC6[j] = curr;
2221                pC7[j] = curr;
2222            }
2223        } // section
2224        } // sections
2225    }
2226    // compute v = Ap
2227    #pragma src parallel sections
2228    {
2229    #pragma src section
2230    {   // send the dot8 vals to accumulator
2231        int i;
2232
2233        for (i=0; i<knz/K; i++) {   // pipeline depth: 63
2234                                    // all k-groups in each row
2235            // grab next k matrix values (a8 values)
2236            split_64to32_flt_flt(kvalL01[i],&a1,&a0); // 2 each
2237            split_64to32_flt_flt(kvalL23[i],&a3,&a2); // unpack
2238            split_64to32_flt_flt(kvalL45[i],&a5,&a4); // from
2239            split_64to32_flt_flt(kvalL67[i],&a7,&a6); // OBM
2240
```

```
2241            // grab next k column pointers (j8 columns)
2242            split_64to32(kcolL01[i],&j1,&j0);          // 2 each
2243            split_64to32(kcolL23[i],&j3,&j2);          // unpack
2244            split_64to32(kcolL45[i],&j5,&j4);          // from
2245            split_64to32(kcolL67[i],&j7,&j6);          // OBM
2246
2247            p0 = pC0[j0];          // use the j8 column pointers
2248            p1 = pC1[j1];          // to grab matching p's (p8)
2249            p2 = pC2[j2];          // remember, we have k
2250            p3 = pC3[j3];          // copies of p
2251            p4 = pC4[j4];          // to avoid memory-based
2252            p5 = pC5[j5];          // loop-carried dependence
2253            p6 = pC6[j6];
2254            p7 = pC7[j7];
2255
2256            // binary tree pipeline for dot8
2257            // recall, this delivers a result
2258            // every clock cycle (after the latency)
2259            //
2260            // a0,p0--->*mn0
2261            //              >+an0
2262            // a1,p1--->*mn1      \
2263            //                     >+an4
2264            // a2,p2--->*mn2      /      \
2265            //              >+an1         \
2266            // a3,p3--->*mn3               \
2267            //                              >+dot8--->
2268            // a4,p4--->*mn4               /
2269            //              >+an2         /
2270            // a5,p5--->*mn5      \      /
2271            //                     >+an5
2272            // a6,p6--->*mn6      /
```

```
2273            //              >+an3
2274            // a7,p7--->*mn7
2275
2276            mn0  = a0 * p0;       // leaf nodes (multiplier stage)
2277            mn1  = a1 * p1;
2278            mn2  = a2 * p2;
2279            mn3  = a3 * p3;
2280            mn4  = a4 * p4;
2281            mn5  = a5 * p5;
2282            mn6  = a6 * p6;
2283            mn7  = a7 * p7;
2284
2285            an0  = mn0 + mn1;     // first adder stage
2286            an1  = mn2 + mn3;
2287            an2  = mn4 + mn5;
2288            an3  = mn6 + mn7;
2289
2290            an4  = an0 + an1;     // second adder stage
2291            an5  = an2 + an3;
2292
2293            dot8 = an4 + an5;     // root node output adder
2294
2295            // stream next dot8 to streaming accumulator
2296            put_stream_flt(&Sva,dot8,1);
2297
2298        } // for i (next k-group)
2299        stream_term(&Sva);  // termination required by accum
2300    }
2301    #pragma src section
2302    {   // send # of dot8's per row to the accumulator
2303        int i;
2304        int curr,prev; // current & previous to calculate counts
```

```
2305        int dot8cnt;   // number of dot8s per row
2306
2307        // note: kptr includes n+1'th item, also
2308        // transfer curr to prev is part of loop control
2309        prev = kptrL[0]; // preload initial prev value
2310        for (i=1; i<=n; prev=curr, i++) {  // pipeline depth: 5
2311                                           // calculate # of
2312            curr = kptrL[i];               // partial dot products
2313            dot8cnt = curr - prev;         // per matrix row, send
2314            put_stream(&Sca,dot8cnt,1);    // to streaming accum
2315        }
2316        stream_term(&Sca);  // termination required by accum
2317    }
2318    #pragma src section
2319    {   // streaming accumulator
2320        stream_fp_accum_strm_counts_32_rr_term(&Sva,&Sca,&Ssa);
2321        //                                        ^     ^    |
2322        // input dot product vals (dot8s) ------+     |    |
2323        // input # of dot8s for each row ------------+    |
2324        // output full dot product stream (dotN) <--------+
2325    }   // section
2326    #pragma src section
2327    {   // finish v = Ap calculation, store in OBCM
2328        float prev,curr;     // last two v[i]'s
2329        int64_t packed;
2330        int i;
2331
2332        for (i=0; i<n; prev=curr,i++) {   // pipeline depth: 100
2333            get_stream_flt(&Ssa,&curr);   // from accumulator
2334            if (i%2) { // odd, so combine
2335                comb_32to64_flt_flt(curr,prev,&packed);
2336             }
```

```
2337                put_stream(&Sv,packed,(i%2));  // when it is odd
2338            }
2339        }   // section
2340            // matrix vector multiply complete: send result back
2341        #pragma src section
2342        {   // DMA stream out v values in packed stream
2343            streamed_dma_gcm(&Sv,
2344              STREAM_TO_PORT,PATH_0,vA,1,(int64_t)vsz);
2345        }   // section
2346        }   // sections
2347    }
```

A.3 Performance Evaluation Application

A.3.1 usec.h

```
2348    // usec.h
2349    // need usec time resolution to measure performance
2350    double usec(void);
```

A.3.2 usec.c

```
2351    #include <sys/time.h>
2352    #include <stdlib.h>
2353    //-------------------------------------------------------------------------
2354    // need microsecond resolution to measure performance
2355    double usec(void) {
2356
2357        struct timeval t0;      // time
2358
2359        gettimeofday(&t0,NULL);
2360
2361        // if we overrun the epoch, boom!
```

```
2362        // we'll assume this seldom happens!!
2363        return((double)(t0.tv_sec*1e6+t0.tv_usec));
2364    }
```

BIBLIOGRAPHY OF LIST OF REFERENCES

Abed, K. H. and G. R. Morris (2009, June). Improving performance of codes with large/irregular stride memory access patterns via high performance reconfigurable computers. In *Proceedings of the High Performance Computing Modernization Program Users Group Conference 2009 (UGC'09)*, San Diego, CA, pp. 422 – 429.

Altera Corp. (2011). FPGA, CPLD, and ASIC from Altera. *www.altera.com*.

Amdahl, G. (1967). Validity of the single processor approach to achieving large-scale computing capabilities.

Axelsson, O. (1996). *Iterative Solution Methods*. Cambridge Univeristy Press.

B. Jacobs, G. G. (2013). Sparse matrix format.

D. Dodson, R. G. and J. Lewis (1991). Sparse extensions to the fortran basic linear algebra subprograms, acm transactions on math software.

Davis, T. and Y. Hu (2011, November). The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software 38*(1), 1 – 25.

Dongerra, J. (2000). Sparse matrix storage formats.

Harkins, J., T. El-Ghazawi, E. El-Araby, and M. Huang (2005, December). Performance of sorting algorithms on the SRC 6 reconfigurable computer. In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology*, Singapore, pp. 295–296.

Hawkins, J. D., A. M. Alfarra, G. R. Morris, and K. H. Abed. Finding the "sweet spot" when mapping scientific kernels onto high performance reconfigurable computers", booktitle =.

Herbordt, M. C., T. V. Court, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello (2007, March). Achieving high performance with FPGA-based computing. *Computer 40*(3), 50–57.

Herbordt, M. C., M. A. Khan, and T. Dean (2009, September). Parallel discrete event simulation of molecular dynamics through event-based decomposition. In *Proceedings of the 20th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, Boston, MA, pp. 129–136.

IEEE (2014). Technical awards, gene amdahl.

J. L. Hennessy and D. A. Patterson (2003). *Computer Architecture: A Quantitative Approach* (Third ed.). San Francisco: Morgan Kaufman.

Jane's Information Group (2011). Tactical reconnaissance and counter-concealment enabled radar (TRACER)(United States). *Jane's Electronic Mission Aircraft*.

Justin L. Rice, K. H. A. and G. R. Morris (2009, June). Design heuristics for mapping floating-point scientific computational kernels onto high performance reconfigurable computers. *Journal of Computers 4*(6), 542 – 553.

Mentor Graphics (2010). DK Design Suite. *www.mentor.com/products/fpga/handel-c/dk-design-suite*.

Morris, G. R. (2006, December). *Mapping Sparse Matrix Scientific Applications onto FPGA-Augmented Reconfigurable Supercomputers*. Ph.D.E.E. dissertation, University of Southern California, Los Angeles, CA, USA.

Morris, G. R. (2013, September). Conjugate gradient. CPE505 –Analysis of algorithms (handout).

Morris, G. R. and K. H. Abed (2013, January). Mapping a Jacobi iterative solver onto a high performance heterogeneous computer. *IEEE Transactions on Parallel and Distributed Systems 24*(1), 85 – 91.

Morris, G. R., R. Y. McGruder, and K. H. Abed (2010, June). Accelerating a sparse matrix iterative solver using a high performance reconfigurable computer. In *Proceedings of the High Performance Computing Modernization Program Users Group Conference 2010 (UGC'10)*, Schaumburg, IL, pp. 517 – 523.

Morris, G. R. and V. K. Prasanna (2006, September). A hybrid approach for accelerating a sparse matrix Jacobi solver using an FPGA-augmented reconfigurable computer. In *Proceedings of the 9th Military and Aerospace Programmable Logic Devices Conference (MAPLD'06)*, Washington, DC.

Morris, G. R. and V. K. Prasanna (2007, March). Sparse matrix computations on reconfigurable hardware. *Computer 40*(3), 58 – 64.

Morris, G. R., A. R. Silas, and K. H. Abed (2012, March). Analytical and measured bandwidth for an FPGA-based processor. In *Proceedings of the IEEE SoutheastCon 2012 (SoutheastCon'12)*, Orlando, FL, pp. 1 – 7.

Nallamuthu, A., M. C. Smith, S. Hampton, P. K. Agarwal, and S. R. Alam (2010). Energy efficient biomolecular simulations with fpga-based reconfigurable computing. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, New York, NY, USA, pp. 83–84. ACM.

NIST (2004, June). Matrix Market. `math.nist.gov/MatrixMarket`.

Paige, C. and M. Saunders (1975, September). Solution of sparse indefinite systems of linear equations. *Society for Industrial and Applied Mathematis 12*(4), 617–629.

Parker, M. (2009). Taking advantage of advances in fpga floating-point ip cores.

Peay, N. S., G. R. Morris, and K. H. Abed (2011, March). Integrating Quartus Wizard-based VHDL floating-point components into a high performance heterogeneous computing environment. In *Proceedings of the IEEE SoutheastCon 2011 (SoutheastCon'11)*, Nashville, TN, pp. 413 – 417.

Rice, J. L., K. H. Abed, and G. R. Morris (2009). Design heuristics for mapping floating-point scientific computational kernels onto high performance reconfigurable computers. *JCP 4*(6), 542–553.

Rice, J. L., K. C. Pace, M. D. Gates, G. R. Morris, and K. H. Abed (2008, April). Reconfigurable computer application design considerations. In *Proceedings of the IEEE SoutheastCon 2008 (SoutheastCon'08)*, Huntsville, AL, pp. 236 – 243.

Roldao, A. and G. A. Constatinides (2010). A high throughput FPGA-based floating point conjugate gradient implementation for dense matrices. *ACM Transactions on Reconfigurable Technology and Systems 3*(1), 1–19.

Saad, Y. (2009). SPARSKIT: A basic tool-kit for sparse matrix computations (version 2). *www-users.cs.umn.edu/~saad/software/SPARSKIT*.

Scipy (2009). Sparse matrix format.

Scrofano, R., M. Gokhale, F. Trouw, and V. Prasanna (2006, April). A hardware/software approach to molecular dynamics on reconfigurable computers. In *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, pp. 23–24.

Song, H. Preconditioning techniques analysis for cg method.

SRC Computers Inc. (2014). Carte Programming Environment. `www.srccomp.com/SoftwareElements.htm`.

SRC Computers, LLC (2010). General purpose reconfigurable computing systems. *www.srccomp.com/products/mapstationworkstations.asp*.

Van der Vorst, H. (2000, January). Krylov Subspace Iteration. *Computing in Science & Engineering 2*(1), 32–37.

Wu, G., Y. Dou, Y. Lei, J. Zhou, M. Wang, and J. Jiang (2009, April). A fine-grained pipelined implementation of the LINPACK benchmark on FPGAs. In *Proceedings of the 17th IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, pp. 183–190.

Xilinx, Inc. (2006). How Xilinx began. In *Xilinx: our history*. www.xilinx.com/company/history.htm.

Xu, N.-Y., X.-F. Cai, R. Gao, L. Zhang, and F.-H. Hsu (2009, January). Fpga acceleration of rankboost in web search engines. *ACM Trans. Reconfigurable Technol. Syst. 1*, 19:1–19:19.

Zhuo, L., G. R. Morris, and V. K. Prasanna (2007, October). High-performance reduction circuits using deeply pipelined operators on fpgas. *IEEE Trans. Parallel Distrib. Syst. 18*, 1377–1392.

Zhuo, L. and V. K. Prasanna (2005a, November). High performance linear algebra operations on reconfigurable systems. In *Proceedings of the ACM/IEEE SuperComputing 2005 Conference*, Seattle, WA, pp. 2–13.

Zhuo, L. and V. K. Prasanna (2005b, February). Sparse matrix-vector multiplication on FPGAs. In *FPGA'05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, pp. 63–74.

VITA

Jamory Dante Hawkins [redacted]. He attended Lanier High School and immediately following graduation, enrolled in Hinds Community college in 2006. On December 12, 2008, Mr. Hawkins received his Associate of Arts degree from Hinds and the following month, in January 2009, enrolled in Jackson State University's Computer Engineering program.

Mr. Hawkins received his Bachelor of Science degree in Computer Engineering on December 8, 2012, continued with his academic goals, and is pursuing a MS degree in Computer Engineering in addition to joining Jackson State University's HPRC research group. During his years of study, he was employed as a graduate research assistant. He has also interned at the U.S. Army Corps of Engineers, Engineer Research and Development Center in Vicksburg, Mississippi. On December 19, 2013, Mr. Hawkins accepted a federal civilian position within the U.S. Department of the Navy's Naval Surface Warfare Center (NSWC) as a Computer Engineer. He is expected to begin his career at the NSWC and receive his Master of Science degree in Computer Engineering in May 2014.